THE OFFICIAL GUIDE

OXFORD PASCAL

ON THE COMMODORE 64

IAN SINCLAIR

Cassell
COMPUTING

OXFORD PASCAL ON THE COMMODORE 64

OXFORD PASCAL ON THE COMMODORE 64

Ian Sinclair
BSc, MIEE

Cassell

Cassell Ltd: 1 St Anne's Road,
Eastbourne, East Sussex, BN21 3UN

British Library Cataloguing in Publication Data

Sinclair, Ian R.
Oxford Pascal on the Commodore 64.—(Cassell computing)
1. Commodore 64 (Computer)—Programming
2. BASIC (Computer program language)
I. Title
001.64'24 QA76.8.C64

ISBN 0-304-31267-3

Typeset by Phoenix Photosetting, Chatham
Printed in Great Britain by Mackays of Chatham

Last digit is print number: 9 8 7 6 5 4 3 2 1

CONTENTS

FOREWORD

At Oxford we have long been aware of the need for a good book on Pascal aimed specifically at the micro user. OCSS now supplies Pascal compilers for both the Commodore 64 and the BBC 'B' computer, and will shortly be producing a version for the Amstrad machine. The Commodore and BBC versions will be the subjects of books by Ian Sinclair under the Cassell Computing imprint.

As compiler writers we are aware that languages require perhaps more in the way of documentation and technical support than any other software. There could scarcely be a better way for the user to get to grips with a Pascal implementation than to work with a text specifically aimed at users of that software. We therefore warmly welcome this book as a valuable aid to the novice and old Pascal hand alike.

Although many Pascal devotees would wish that it were not so, it is a fact that a large proportion of the new Pascal public originally cut their programming teeth on BASIC. Ian Sinclair's book does not fall into the trap of merely showing how BASIC programming techniques may be translated into Pascal. Although he does draw on any knowledge of BASIC that the reader may have, he is always at pains to promote the spirit of Pascal and to inculcate Pascal thinking in his readers.

It is now widely accepted in the educational community that learning Pascal is one of the most cost-effective ways in which students of computer science can use their time. Although Pascal is small compared with other Algol-type languages, there are few modern programming ideas which cannot be illustrated in it. Pascal enforces strong data typing and supports such features as recursion, abstract data types, procedures, static nesting, linked lists and case structures.

One of the chief merits of the book is the inclusion of a wide range of example programs. Among these is a useful set of string routines which, when included in Pascal programs, give many of the facilities familiar to users of UCSD

Pascal. Somewhat at odds with the strictest view of high-level programming is the facility present in Oxford Pascal to incorporate machine code routines as external procedures in Pascal programs. It would be nice, of course, if we could all live entirely in a Pascal world and never be forced to leave it for the vulgarities of bits, bytes and machine registers. In the world of micros, however, this is not possible, and not worth attempting. Programmers will always need to resort to machine code from time to time, and Mr Sinclair explains just how this may be done.

Overall then, Oxford Pascal is a practical guide to Pascal the language and, in particular, to the Oxford Pascal implementation by OCSS.

Finally, perhaps I should mention that Oxford Pascal (the software) can be obtained directly from OCSS by filling in and sending the order form at the back of this book.

TONY WILKES
Technical Director
Oxford Computer Systems (Software) Ltd

PREFACE

Though Pascal has been used for a considerable time as a language for teaching computer programming, and as a working language, it has never displaced BASIC as a language for microcomputers. This has been due to several reasons, some of which are no longer valid. One reason has been that BASIC, because it allowed direct keyboard commands, was an easier language to learn. This is certainly true, but it is also true that it is easier to learn to program badly in BASIC than to program well. Another restriction was that many Pascal compilers did not produce machine code which could then be recorded and used on any suitable machine. It was common to find that Pascal programs could be run only on machines which were equipped with a set of Pascal routines. This is still true for the cassette-based Pascal systems, but not for disk systems. The most vexing problem was that Pascal was simply not available for machines with small memory sizes. The result was that this useful, fascinating, and *simple* language was confined to programmers who had mini-computers rather than microcomputers.

The arrival of Oxford Pascal has changed the outlook completely. The version around which this book is written has been designed for the Commodore 64, a machine which has been sold in very large numbers all around the world. The Commodore 64 was not blessed with a version of BASIC which allowed a BASIC programmer to make use of its considerable facilities, and for spectacular sound and graphics programs, the use of machine code was necessary. Now, using Oxford Pascal, the programmer can operate with a high-level language which includes instructions for the high-resolution graphics and sound, and which compiles to very fast-running machine code. The serious programmer who uses disks can also produce Pascal programs in machine code which will run on C-64s that are not equipped with Oxford Pascal. This language, then, provides all that the C-64 BASIC lacks, and much more.

I have assumed that the reader of this book will have programmed in BASIC.

Since the book is written for the C-64 user, this seems to be a reasonable assumption, and it provides a basis of comparison between BASIC and Pascal. In some ways, it is easier to learn a new computing language if you have previously learned any other computing language. This is not always true when the first language is BASIC, however, because Pascal offers much that can only be accomplished with a lot of tortuous effort in BASIC. Very often, a BASIC programmer is tied to BASIC methods, and cannot see that there are simpler ways open to him/her when using another language. I have taken every opportunity to point out the new ways in this book. I have also included many reminders about old BASIC habits that must be abandoned when you learn to program in Pascal.

Because the way that you write Pascal programs is very closely tied to the way that you design a program, I have linked these topics together rather than trying to deal with the design of programs separately. The book has been written entirely around the conventional 'top-down' programming method, and the programming has been approached in the same way. The system which was used was a Commodore 64 with the 1541 disk drive, but the book is equally suitable for the user who has bought Oxford Pascal on cassette. Where I have used commands that are peculiar to the disk system, I have pointed this out, and most of these specialised topics are dealt with in Chapter 9. My Oxford Pascal disk was a stock item, purchased by mail, and not a pre-production special. The reader should find, then, that there are no differences between the Pascal of my descriptions and the Pascal that he/she has bought. The programs that appear in this book have been taken from listings which were produced by an Epson printer reading the Pascal program direct from disk. The listings in the book are therefore exactly of the programs which I used. Because of the peculiarities of C-64 ASCII codes, the character which is shown as the underline on the printout from a Commodore printer appears as the dollar sign in these listings. Also, the up-arrow appears as an inverted 'v', as it does on many keyboards of other machines.

Finally, I am very grateful to Oxford Computer Systems for this excellent version of Pascal, which has breathed new life into an old machine. I am also most grateful to Chris Coyer, of Holt-Saunders, who warmed to the prospects of Pascal, and commissioned this book.

IAN R. SINCLAIR
March 1985

1

INTRODUCTION TO PASCAL

WHY PASCAL?

If you program in BASIC and find the language perfectly adequate, it's reasonable to ask 'Why Pascal?'. Certainly, if you find the BASIC of the Commodore 64 completely suited for what you want to do, you probably don't need Pascal. On the other hand, it's much more likely that you have found a lot of items for which BASIC, and particularly Commodore 64 BASIC, is not well suited. One obvious set of such items on the Commodore 64 is sound and high resolution graphics, which require the use of POKE to memory. Others are less obvious, unless you are trying to write business software that requires the use of lists, or games software which uses sets of items. BASIC can cope with these needs only in roundabout ways, and if you have only ever programmed in BASIC, you simply don't realise how much effort you are having to make for what are quite simple actions in other languages.

Another reason for using Pascal, of course, is speed. BASIC is an interpreted language. This means that every instruction in a BASIC program is carried out as the machine comes to it. A computer carries out these instructions by converting them into machine code and running the machine code. An interpreter takes the instructions one by one, finds the machine code for an instruction, runs that instruction, then looks for the next. By contrast, a compiler deals with the whole of a program and converts it into one piece of machine code. This code can then be recorded and run. The advantage of this is that all of the looking-up and converting steps are done once, in the action which is called 'compiling', and this compiled code can run fast, almost as fast as machine code which has been written using an assembler. When you use an interpreted language, the interpretation has to be done for each step and each time the program is run. For example, suppose that your program consists of a loop which prints a string variable fifty times. An interpreter would look for the

machine code for the print operation, find the address and length of the string of characters, and carry out the action fifty times. All of the looking up is repeated on each pass through the loop. A compiler, by contrast, would generate machine code which carried out the print action in a loop. This generation would be done once, when compiling, and because the looking up actions do not have to be done again, the result is very much faster. The speed of a compiled language may be important for you if you want to write either business or games software.

Yet another reason for using Pascal is that the language is much closer to the languages that are used on larger machines. It is also 'portable' in the sense that a program which is written in Pascal for one machine will probably work with another machine. This assumes only that there are no instructions which are peculiar to one machine, and Pascal is particularly good in this respect. Above all, though, Pascal is a language that forces you to think carefully about what you want a program to do. It *is* possible to write poorly-designed and sloppy programs in Pascal. It is also possible to write neat and tidy programs in BASIC. In the normal run of events, however, both of these are unusual.

THE ORDER OF THINGS

Pascal is not simply a compiled language, it is a language which has to be written so as to make it *easy* to compile. This means that your instructions for a Pascal program should be written in the same order as that in which the compiler will convert it into machine code, or as near as possible to the same order. This point is very important, and it is the reason for the order of items in a Pascal program, an order which will look strange to anyone who has been brought up on BASIC. You should think of the compiler as reading the program from start to finish. If the program can be read without any need to 'look forward', meaning that nothing appears in the main program that has not previously been defined, then the program is probably a good one.

This idea is not one that appears much in BASIC. A few machines which use Microsoft BASIC (such as the MSX machines) use instructions such as DEFINT. This allows you to define how letters will be used in the program. For example, DEFINT A-D means that any variable name which starts with the letters A, B, C, or D will be an integer variable. This means that you no longer have to mark integer variables, like A%, BY%, COWS% and so on, to show that they are integers. The DEFINT statement at the start of the program has done this for you, and it's also possible to define string or real-number variables in a similar way. None of this can be done with the BASIC of the C-64.

In Pascal, however, this idea of declaring how names will be used in advance is all-important. Note, too, that I said 'names'. In Commodore 64 BASIC, you are probably used to working with variable 'names' like A, BE$ and so on. Pascal allows you to use realistic names rather than just letters singly or in pairs.

You must, however, define what type of item each name will be used for. This allows the compiler to prepare for each variable that will be used by making the correct allocation of memory. You cannot write statements like

```
NAME="SINCLAIR"
```

in Pascal unless you have, earlier in the program, declared that NAME is a variable that will be used for a string of at least eight letters. You can't make this declaration later, because the compiler will halt when it finds a name used that it has no record of. You cannot use a name unless you have defined the name. The definition does not necessarily need to be placed at the start of the program, but it certainly must come before you attempt to use the name. So that I don't have to use long-winded phrases each time I remind you of this idea, I'll give it its correct name from now on—it's called *pre-definition*.

PROGRAM STRUCTURE

The structure of a program means how it is arranged. Some BASIC programs are about as structured as the path of a fly. Others are neatly arranged with a simple main core program which calls subroutines to perform the actions. If you have written programs of the core-and-subroutine type, then it's likely that you'll take well to Pascal. If your programs have been of the 'fly-track' variety, you will have problems. Pascal forces you to have some structure about your programs, and the type of structure is one that is far removed from BASIC. For example, in BASIC, you might call up a subroutine in a line like

```
220 GOSUB 5000:PRINT A;
```

which carries out a subroutine, prints the value of a number, and keeps the printing in the same line.

This could never be mistaken for a Pascal line. For one thing, Pascal lines aren't normally numbered, although Oxford Pascal for the Commodore 64 uses line numbering. If you write the lines in the correct order, the order that the compiler will deal with them in, there's no need for line numbers. The second point is that the subroutine starts somewhere later in the program, at line 5000. We can't have such a thing in Pascal, because the compiler must read the subroutine before it is to be used. Instead of using a subroutine which is called by its line number, Pascal uses a procedure which is called by using its name. Users of the BBC Micro and the QL are familiar with this idea, but it will be new to Commodore 64 BASIC programmers. Even the instruction PRINT is not used in Pascal, and the semicolon does not mean 'don't take a new line'. Who said that knowing one computer language would prepare you for another?

The point that is really important here, though, is *order*. In BASIC, you can write a core program which has calls to subroutines. It won't run correctly until the subroutines have been entered, but you can place the subroutines anywhere you like in the program. Defined functions in Commodore 64 BASIC

behave much more like such functions in Pascal. If you have a function which, for example, multiplies a price by 0.15 so as to work out the amount of VAT, you must define it before you use it. You cannot have lines in BASIC such as

```
10 INPUT "BASE PRICE";X
20 PRINT"VAT IS ";FNA(X)
30 DEF FNA(S)=S*.15
```

because the BASIC interpreter cannot look ahead for FNA, only back. The similarity here is that a defined function in Commodore 64 BASIC is called into action by using its *name*, FNA in this example, rather than by using a line number. In a Pascal program all defined functions and procedures must be defined before they can be used. 'Before' in this sense means earlier in the program, because the compiler starts at the beginning and works through the program statements in order. Unlike a well-structured BASIC program then, which might consist of a core program of perhaps ten to a hundred lines of main program followed by subroutines, a Pascal program is written in the reverse order. This means that details, such as declaring variable types, actions of procedures and defined functions, all come at the start of the program, and the program itself comes last of all. This is an upside-down arrangement as far as the programmer is concerned, but it's the perfectly logical arrangement from the point of view of the compiler. This doesn't make writing of programs any more difficult; it merely forces us to arrange our programs rather differently. Any modern version of Pascal can be expected to possess a good editing system, so it's no more difficult to add statements at the start of a program than at the end. If you have programmed for a long time in BASIC, you might feel rather lost without line numbers, and Oxford Pascal therefore allows you the use of line numbers. This is shown only in the first of the examples in the manual, but you can use line numbers just as you do in BASIC, and the line numbering is automatic. You can start by typing any line number, 100 perhaps, and entering the first item of the program. When you press RETURN, the new line number 110 (in this example) will appear, and so on. You can also use the AUTO command if you don't want the lines to run in tens. For example, AUTO5 (RETURN) will cause the line number to change in steps of five once you have typed the first line number that you want.

Unlike many versions of Pascal, the line numbering is important in Oxford Pascal. If you try to enter a statement without a line number, you will get a ? SYNTAX ERROR message. The line numbers are also very useful if you want to list selectively, or delete some lines. You can still use all the normal screen-editing methods of the C-64. The line numbers, however, are not recorded with the program, and you can find that your program has an entirely different set of line numbers when you replay it. The examples in this book have been printed along with line numbers. This makes it much easier to refer to lines by number, and for you to check your programs.

The use of line numbering just like BASIC makes it much easier for you to have second thoughts. Suppose, for example, that you always use line numbers

from 1000 up for your main programs. You can then keep line numbers 100 up for functions and procedures, and line numbers from 10 up for variable declarations. These allocations will certainly be sufficient for your early programs. If you find that you are running out of numbers, then Oxford Pascal comes to your rescue with facilities that you don't normally have on the C-64. You can, for example, renumber lines with a *higher* starting number. If you find that you are writing so many function and procedure lines that you will need to use lines from 1000, then you can renumber all the existing lines of the main program by typing NUMBER1000,10000,10. This will renumber line 1000 as 10000, and the following line numbers will step in tens. You can also delete lines. Typing DELETE 100–200 will delete all lines between, and including, 100 and 200. DELETE typed by itself is equivalent to NEW.

In addition to these useful editing commands which are missing from the BASIC of the C-64, Oxford Pascal allows you a couple of commands which are used in word-processing programs. One is FIND, which is used to find any string, such as a name. The command FIND/COPY/ (RETURN) will list on your screen (or printer) each line which contains this word COPY. The mark that is used before and after the word that you are looking for need not be a slash-mark. It can be any mark, such as !, #, $, %, & which is not part of the string that you are looking for. The other action of this type is CHANGE. This can be used to change one string into another. For example, CHANGE/COPY/TURN/ (RETURN) will list all the lines that contain COPY and show this word changed to TURN. You can also make FIND and CHANGE work on a specified line or a range of lines, rather than on the whole program.

RESIDENT AND DISK COMPILING

Oxford Pascal is sold in two versions, cassette and disk. When you use the cassette version, the compiler program is held in the memory of the Commodore 64 for all the time that you need to use it. In addition, the memory contains a form of interpreter for a partly-compiled code. This is necessary, because a cassette system does not allow a program to be loaded so quickly or easily as a cassette program. This way of using Oxford Pascal is called the 'resident mode', meaning that the compiler resides in the memory. The disk version of Oxford Pascal also allows resident mode to be used. This mode is satisfactory for learning to use Pascal, and for most programming purposes. It *is*, however, slightly restricted. The main restriction, as far as the keen programmer is concerned, is that a program prepared in resident mode must be compiled each time you want to use it. For a program which might take a long time to execute anyhow (like a game which can be played for hours), this is no handicap; it simply means that you have to wait until the program compiles before you can use it. Another disadvantage, however, is that only a Commodore 64 which has Oxford Pascal loaded into it can load, compile and run a program which has

been created this way. This is because resident mode does not record or replay machine code. All that is recorded is a serial file of strings, the instructions of the Pascal program. These have to be read and compiled before running is possible. Disk users can use the alternative disk compiler. This loads the compiler into the memory only when it is needed. The program can be read from disk, compiled, and the resulting machine code stored back on the disk. This allows you to run a Pascal program after loading the run-code interpreter. The other option is to convert the program entirely into a version that contains all the Pascal routines. Once this has been done, the code can be read from the disk by any C-64, and run. In this way, a program which you have written in Pascal can be used by any C-64, even one which is not supplied with Oxford Pascal. In addition, disk compiling offers a number of useful commands which cannot be made available on the resident version.

Throughout this book, the examples will be usable on either resident or disk mode. At various points, I shall mention commands that are available to disk users, mainly because I believe that most programmers in Pascal will be using a disk system, as I am. Nevertheless, if you are learning about Oxford Pascal with only a cassette system, you will be able to use all of the examples which are for resident mode. You may encounter problems if you intend to use a printer which is not of Commodore manufacture. Many Commodore 64 users have attached Commodore 64 machines to printers like the Epson RX80, in order to take advantage of the better print quality of these machines. This can be done if suitable hardware connectors are available, and with a software 'printer driver' program. The use of such a printer drive conflicts with Oxford Pascal, unfortunately, because both are trying to use the same parts of memory. So that I can use the Epson printer with Pascal programs in this book, I have written a short program in BASIC which reads the Pascal programs as serial files, adds line numbers, and prints the result. The program is illustrated in Figure 1.1. Like the other listings in this book, it has been printed with the machine set for 40 characters per line, so as to show how the listing will appear on the Commodore 64 screen. This allows recorded programs to be printed later, when Pascal is not in use. Note, however, that the underscore character, which is obtained by using the C= key and @ key together, shows in listings as the $ sign. The programs that are illustrated all use upper-case letters. You can use either lower or upper-case letters on the screen, using the C= and SHIFT keys to switch cases. If you use a Commodore VIC-1515 printer, however, you will find that listings in upper-case are printed as a set of graphics symbols, but listings in lower-case are printed in upper-case. The command POKE 49153,1 should allow the Commodore printer to show upper-case commands correctly, but I found that my printer always hung up midway through a listing when the Oxford Pascal DUMP command was in use. For that reason, it's better to avoid the DUMP command of Oxford Pascal. The Epson printer will show the listings in the case that has been used. I have written all programs in upper-case, with a Commodore printer to check the listings, but with the Epson to make the final copy. In

```
2 OPEN6,6
3 PRINT#6,CHR$(27);CHR$(69)
4 PRINT#6,CHR$(27);"0";CHR$(40)
5 CLOSE6
10 PRINT"FILENAME":INPUT F$:N%=10
20 OPEN 3,8,3,"0:"+F$+",S,R"
30 OPEN 1,6
31 REM ALTER FOR PARALLEL
40 S$=""
42 GET#3,A$:IF ST<>0 THEN 70
44 S$=S$+A$:IF A$<>CHR$(13)THEN 42
50 PRINT#1,N%;" ";S$;
60 N%=N%+10:GOTO 40
70 N%=N%+10:CLOSE 3:PRINT#1,N%;" ";S$
80 CLOSE 1
90 GOTO 10
```

Figure 1.1 A program in BASIC for printing out the Pascal source files. Line 30 is written for an Epson printer.

the text of the book, also, I have shown all program instructions and command words in upper-case and they are also in a different typeface to distinguish them from the rest of the text.

SOURCE CODE, P-CODE AND OBJECT CODE

The use of a compiler brings you into contact with some phrases that you will not have met previously if you have programmed only in BASIC. One such pair of phrases is *source code* and *object code*. The source code for a compiler is the list of instructions, the program that you type. This is stored in memory as a set of ASCII codes, and it can also be recorded in the same coding, as a serial file. This file can be read by a BASIC program, as Figure 1.1 illustrates. Unlike a program in BASIC, however, a Pascal source code program cannot be run as it is; it has to be compiled first. Compiling results in another lot of code, the *p-code*. This is a mid-way stage between source code and true machine code, and it has to be interpreted in order to be run. This makes it slower than a Pascal which has been compiled to true object code, and it also requires a p-code interpreter to be present in the memory of any machine that is used. When you use Oxford Pascal in the resident mode, you will compile to p-code, which is then interpreted by another part of the program. When you use disk mode, the source code is compiled to p-code, which is then recorded. Before this p-code can be replayed and run, the p-code interpreter (also called the 'run-time system') has to be loaded from disk.

The ultimate compilation is to machine code (also called 'object code') a set of number codes which operate directly on the 6510 microprocessor of the C-64. It is this set of codes which can be recorded, and which runs so quickly when we can use disk mode. As we noted earlier, if you use resident mode of Oxford Pascal, each run requires the source code to be compiled afresh to p-code and then run with the aid of the p-code interpreter. With disk mode, the compilation can be done once and the object code recorded for use later. When the p-code is combined with the run-time system, the resulting code can be loaded like a BASIC program, but runs like a machine-code program.

OXFORD PASCAL SUMMARISED

Unless you have used other varieties of Pascal, the advantages of Oxford Pascal are not necessarily clear to you, even after reading the manual. The main point is that Oxford Pascal is a version which corresponds very closely to international standards. You should never have to relearn your Pascal, because there are not the 57 varieties of Pascal that you find with BASIC. If you change your computer for a more modern design, then you will find that you can use Pascal on the new machine as readily as you did on the old. This is particularly true if Oxford Pascal is also available for your new machine.

Having praised the advantages of standardisation, however, there are also some non-standard advantages. The Commodore 64 is a machine which has quite superb capabilities for graphics and sound, better than those of later Commodore machines. Oxford Pascal has been extended to allow you to control these features of the C-64, and obviously these features would not necessarily be available on other computers. These instructions avoid all the tedious POKE commands that were necessary with Commodore 64 BASIC, and for graphics, they allow much more control than was possible before. In particular, Oxford Pascal features a PLOT command which will draw or clear a line, and fill or clear an area. You can also use commands such as PAPER and INK, which are found in the BASIC of many modern computers. There are also provisions for using hex numbers, writing in assembler language, and placing characters directly into screen memory (using the VDU command).

The BASIC commands that are listed below can also be used *directly* from the keyboard.

PRINT (or ?)—print on screen
PRINT#—print to printer or disk
OPEN—open file or channel
CLOSE—close file or channel
CMD—transfer screen listing to printer
LET—assign value
LOAD—load disk or cassette file
PEEK—look at memory

POKE—alter memory
SYS—start machine code program
Note: These *cannot* be used in Pascal programs, only as direct commands from the keyboard.

In particular, you may prefer to make use of OPEN3,4 followed by CMD3 and LIST to list your Pascal programs. I found that the use of DUMP, which is the recommended command for listing to the printer, was not always reliable. DUMP would normally operate on a program which had been written in lowercase, recorded, and then replayed. It would often cause the printer to hang up if used on a program which had just been typed. This is not a serious problem, because the VIC-1515 printer can be switched off, allowing the listing to continue.

The use of line numbers when entering Pascal statements makes editing particularly easy, especially when you can make full use of the familiar editing system of your C-64. This makes learning Oxford Pascal very much easier, because you do not have to struggle with an unfamiliar editing system at a time when you are likely to make a large number of mistakes. You also have command over the Commodore 64 real-time clock, and you can load and display the disk directory as usual. Remember, however, that loading the disk directory will erase any program which is in the memory.

For disk users who choose to use the disk mode, there are many advantages in Oxford Pascal. Of these, the main advantage is that disk files can be used in programs, and that all the file types of standard Pascal can be used. Programs can be chained, so allowing you to create and run very large programs. Chaining means that at some point in the running of one program, another program can be called up in the same way as we call up subroutines. If the programs are written correctly, the same variables will be used, and the first program can resume after the chained section has run. It is possible in this way to run a program which would be too large to fit into the memory of the computer. Another facility is linking, which means that programs in object code can be joined into one longer program. Once again, there are restrictions about how the programs are written, but this is yet another way in which the creation of programs can be made much easier when you work with Oxford Pascal. The special features of disk mode are dealt within Chapter 9.

2

STARTING PASCAL PROGRAMS

LOADING IN

The Pascal disk is loaded into your Commodore 64 in much the same way as any other disk which contains machine code. The disk contains several sections of code, and when you use LOAD"*",8 or LOAD"PASCAL",8, then what is loaded is a one-line BASIC program which simply brings in the main machine code section. When this one-liner loads, you have to RUN it to load in the main section. If you prefer, you can simply clear the screen and then type LOAD"LD",8,1 to get into the main section right away.

In the middle of normal loading, you will get a message to the effect that the program is loading. On a bad copy, this message appears at the end of loading, and the machine then locks up in the familiar way. On a good copy, the disk drive continues a lot of activity well after the loading message has appeared. Eventually, you will see the ready prompt appear, and you are about to start working with Pascal. You should now take the Oxford Pascal disk out of the drive and put it in a *very* safe place. If, of course, you have used Disc Disector V2.0 to make a fast-loading back-up, you will have no problems about disk security, nor so much time to worry while the disk is loading!

FIRST STEPS

All Pascal programs can be constructed in the same way, and though you can leave out some steps in Oxford Pascal, it's advisable to keep to the rules of standard Pascal. If you do, it's much easier to write Pascal programs for practically any machine. In addition, it helps a lot if you write programs the way that you ought to design them, outline first and details later. The outline of a Pascal program then, is

```
10 PROGRAM Title;
20 BEGIN;
30 (*Statements*);
40 END.
```

and we now have to take a look at this to decide what is important in these few lines. One thing which is very important is the way that the semicolon is used. In BASIC, the semicolon is used to show that printing is to be kept on the same line. In Pascal, the semicolon is used as a separator, showing that you have reached the end of a statement but that there is more computing to be done. The rule about the semicolon can be written very simply—the semicolon marks the end of a statement. Until you have had rather more experience with Pascal, however, you don't necessarily know where a statement ends. A good guide, however, is always to put a semicolon at the end of a line unless you are quite certain that there should not be one.

In the example of program outline, the first line consists of the keyword PROGRAM and a name 'Title'. A keyword in Pascal is rather like a keyword in BASIC; it is reserved for a special purpose and you can't use it for anything else. Keywords must be correctly spelled, otherwise a 'syntax error' will be announced when you try to compile. The keywords of Oxford Pascal are as follows. These cannot be changed and must not be used as identifiers (variable names). Ignore the brief notes until you have read the remainder of the book.

AND—joins two conditions
ARRAY—defines type of array
BEGIN—indicates start of program or sub-program
CASE—used for menu selection
CONST—identifies a constant or list of constants
DIV—gives result of integer division
DO—marks start of a list of actions
DOWNTO—used to decrement counter in FOR loop
ELSE—marks alternative action in IF test
END—marks end of program or sub-program
FILE—defines a variable type as a disk file
FOR—marks start of FOR loop
FUNCTION—marks definition of function
GOTO—used with label number to jump to a statement
IF—tests value of variable
IN—checks whether a value is included in a range
LABEL—used to mark destination of GOTO
MOD—gives remainder of integer division
NIL—sets variable to zero
NOT—inverts value
OF—used in CASE and other selection places
OR—alternative test
PACKED—used to store arrays so as to take up minimum memory

PROCEDURE—marks start of procedure
PROGRAM—used in first line to identify program name
RECORD—defines record type and contents
REPEAT—marks start of REPEAT...UNTIL loop
SET—defines a set of objects
THEN—defines action after IF test
TO—gives end of range of FOR loop
TYPE—defines type of variable
UNTIL—marks end of REPEAT...UNTIL loop
VAR—start of variable definition
WHILE—start of loop; must be followed by a test
WITH—indicates actions to be done with parts of a record

Most of these will be found on other versions of Pascal, but other versions, notably the Pascal for the IBM PC, may have a few more keywords. PROGRAM is a keyword which is reserved for the purpose of identifying your program. You are not forced to have each program begin in this way in Oxford Pascal, but it's a good habit to get into.

The word Title, which follows PROGRAM is your name for your program. This is an example of a word which is an *identifier*. In BASIC, the only identifiers that you use are file names and names for variables. Pascal uses a lot more identifiers, and they are used in much more interesting ways. In this case, the word Title identifies the program, but it's not particularly useful otherwise. You can change this name to anything else that you like, subject to some rules. The rules are that this identifier must start with a letter, with upper-case and lower-case being treated as identical. You can then follow this letter with other letters or with digits, but no blanks or punctuation marks. The only character that is allowed, apart from letters and digits, is the underscore (_), which you get by pressing the C= and @ keys at the same time. Unfortunately, this prints in my listings as a dollar sign. You can use names of more than eight characters, but only the first eight characters will count. This gives you a lot more choice about things like variable names than you have in BASIC.

You may remember that in Commodore 64 BASIC, there are some variable names that you cannot use, such as ST, TI and TI$. Oxford Pascal, like other Pascals, has some identifier names which are already allocated. These are as follows:

ABS	EOLN	NEW	READ	SQRT
ARCTAN	EXP	ODD	READLN	SUCC
BOOLEAN	FALSE	ORD	REAL	TEXT
CHAR	GET	OUTPUT	RESET	TRUE
CHR	INTEGER	PACK	REWRITE	TRUNC
COS	INPUT	PAGE	ROUND	UNPACK
DISPOSE	IN	PRED	SIN	WRITE
EOF	MAXINT	PUT	SQR	WRITELN

When you look at this list, you might think that these were another set of

reserved words, as many of them would be in BASIC. The difference is important. All of these identifier names can be used by you for something else if that's what you want. If, for example, you want to call your program ABS or COS or GET then you can do so. You would be foolish to do this, because by changing the meaning of these names, you are losing the use of some action that you might need, and you should try to avoid doing so, but you will not cause any error message. The difference is important, because if you try to use a reserved word for anything else, the error will be signalled; if you use one of the 'predefined' identifier words, there's no error. Some of these keywords, such as PACK and UNPACK, are very rarely used.

Following the PROGRAM and the identifier Title, there is a semicolon. This must be present, and if you omit it you will see a whole set of error messages when you try to compile. One of the odd features of the Pascal compiler is that one error can generate a lot of messages, many of which will refer to other lines. That's because an error early on in a program will upset the action of the compiler on later parts. Don't be downhearted, then, if you find that you seem to have an error in each and every line of your program. It's just as likely that there is just one error, in the first line! Oxford Pascal will allow you to omit the whole of this first line if you like. You can't, however, omit the next one. Where a program begins has to be marked by the reserved word BEGIN. The semicolon which follows BEGIN is not essential, but it doesn't do any harm. In this very simple program, BEGIN comes immediately following the program name, but this is unusual. Normally, you would find BEGIN some way down the listing. This is because BEGIN marks the start of a piece of program, and before you can have a piece of program, there must be actions like naming variables. This is something that we'll look at very shortly.

The next line is where we would expect the program to do something. Instead, all that we have is (*Statements*). The combination of the brackets and the asterisk has the same effect as REM in BASIC. It marks a piece of the program which is just a reminder to the programmer. Unlike BASIC, you have to mark both the beginning and the end of the remark. In addition, a reminder in Pascal does not slow down the program in the same way as a REM in BASIC does. This is because the compiler ignores the reminder and no code is generated. In BASIC, a REM still has to be looked at each time the program runs, just to read the code that means REM. Finally, the Pascal program ends with END, *and a full stop*. The full stop is very important. The word END can be used in many places in a Pascal program. This is because each section of a program has a beginning and an end, and the words BEGIN and END are used to mark them. Something more is needed, therefore, to mark the end of the *entire* program, and END., with a full stop, is that method. If you miss out the full stop, then the compiler looks for more lines of the program, and then issues another error message when it can't find more.

You can now check the sample program, and compile it. This is done by typing L (RETURN). If you have made no mistakes, you will see the message

COMPILING followed by the lines of the program in order, then the welcome messages 0 ERROR(S) and COMPILATION COMPLETE. If you do find errors in this example, then it's nearly always going to be omission of a semicolon. You can then run the program by typing R (or RUN) and (RETURN). You can also omit the L step, and simply type R to compile *and* run. This is less interesting, because you don't see the lines of the program appear as each is compiled and passed as correct.

OUTPUT TO THE SCREEN

You have probably already noticed that Pascal does not have a reserved word PRINT. There is, in fact, no reserved word for the action of putting something on the screen. This action is one which carries an identifier name rather than being one of the reserved name actions. The identifier words that you need are WRITE and WRITELN. The difference between the two is that WRITE will put something on the screen, and wait for something else on the same line (like using a semicolon in BASIC). WRITELN, by contrast, will place something on the screen and then take a new line ready for whatever comes next. What is 'written' on the screen in this way can be a number or it can be text. Anything that is put on to the screen in this form using WRITE or WRITELN is called a 'text file'. The simplest possible examples of either numbers or text look sufficiently like BASIC to be easily understood when you are reading a Pascal program. Figure 2.1 shows an example, which you can type, compile and run.

```
10  PROGRAM P2N3;
20  BEGIN;
30  WRITELN('WORDS');
40  WRITELN(5+3*2);
60  END.
```

Figure 2.1 A simple Pascal program which prints on to the screen. Note the use of brackets and the apostrophe.

In this example, the actions are of writing a word and performing a piece of arithmetic. The writing of a word is rather different to the PRINT"WORDS" that would be used in BASIC. The word has to be placed between *single* quotes, and also has to be placed between brackets. The semicolon at the end of the line has nothing to do with the printing action, remember; it's just the signal to the compiler that there is more to come. In the next line, what is written is still placed between brackets, but there are no quotes. The arithmetic result is printed out, just as it would be by PRINT5+3*2 in BASIC. As in BASIC, the multiplication is carried out before the addition, so that the result is 11, not 16. Here again a semicolon has been used to make the compiler move to the next line, and this is also one of the places where you don't have to use a semicolon. Once again, it's easier to use it. A good rule, in fact, is to use a semi-

colon just before you press RETURN. We look, as we come to them, at the places where a semicolon *must not* be used.

When your program is compiled and will run, it's a good idea to check recording. To record your program on disk, type PUT0:P2N3, then (RETURN). The drive number is needed if you have not recorded a program since you switched on—you will get a Disk not ready message if you omit it. For cassette saving, see your Oxford Pascal manual. When the program has been saved, you can load it with GET. If you have only one drive, you don't need to specify drive number with GET, but you must use the correct file name. You will find that the program appears renumbered when you list it, with numbers starting at 1000. You can use LOAD for the disk directory (remember that this will wipe out any program from the memory), but not for these Pascal programs. You will see from the disk directory that your Pascal program has been recorded as a sequential file. This file is called the 'source-code'. Until this source code has been compiled it is just a file of ASCII codes, nothing more. Once compiled, it is object code, closer to machine code and quite different in action. The most important difference from your point of view is that the source code can be read, edited, and easily understood. The object code has no meaning unless you know about machine code, is very difficult to edit, and can be recorded only when you are using the disk compiler system. The routine for disk mode compiling is summarised as follows. Do not attempt this until you have gained considerable practice with the resident compiler. Be careful when you use LOCATE that you specify a filename for the machine code file which is not already in use.

1. Load the OP program, and type DISK (RETURN).
2. Type your program (source file) and save it on disk, noting the filename.
3. Put the system disk into the drive.
4. Type COMP filename,L. No markers are needed around the filename, but there must be a space between COMP and the filename. The L gives listing on screen, and other options are listed in the manual.
5. Wait for the message on the screen, and then remove the system disk and insert your program (source-file) disk. Press RETURN. Watch the compilation. If there is any error, note the error message(s). You will have to return to your source file to correct the errors. If there are no errors, a p-code program will be saved on the disk. This will be called filename .OBJ—that is, your filename with .OBJ following it. This is the program that will run.
6. You can now run the program by putting in the system disk and typing EX filename (RETURN). Wait until the Pascal run code has loaded from the system disk, and when you are prompted by the message on the screen, take out the system disk and put in the program disk. Press RETURN, and your program will run.
7. If you want to create a machine code program which will run in any C-64, put in the system disk, and type LOCATE 0:SYSFILE=filename. Your

program must have been compiled, so that there is a file called filename.OBJ on the program disk. The name SYSFILE means any different filename, but it's a good idea to make it start with SYS.

8. When prompted, remove the system disk and put in the program disk. Press RETURN and wait. This takes a long time! When it's ended, the program disk will contain a long file called SYSFILE (or whatever name you used). This consists of one line of BASIC (SYS followed by a number) and a lot of machine code.

Whether you use disks or not, there is no point in using disk mode for short programs. This is because even a short program requires a large amount of code. The example of Figure 2.1 requires 50 blocks of disk storage when it is compiled to a combined BASIC and machine code program which will run independently. The object code (a form of intermediate p-code) is shorter, only four blocks, but the source code fills only part of one block. The reason for the difference is that when you compile and run in one operation, most of the code is already in memory, read from the system disk. When you use the disk mode to create a program which will run on any Commodore 64, it must include all of the code, the 'run-time system', that would normally be held in memory after using the system disk. The consolation is that longer programs do not necessarily need all that much more code. From now on, we will concentrate on examples which use resident mode.

USING CONSTANTS

When you use a constant, like 3.1416, in BASIC, you are always advised to assign the value to a variable. The reason is that this avoids the BASIC interpreter having to convert the ASCII codes for the number into number-variable form each time the number is used. One of the things that you have to get used to with Pascal is that it has a lot more different types of data, and one of these is a constant, which uses the reserved name CONST. The syntax is

```
CONST PI=3.1416;
```

using the reserved word CONST followed by the assignment of the quantity 3.1416 to the name PI. This can be done on more than one line if you like—but don't forget the semicolons. You can, for example, have lines such as

```
20 CONST PI=3.1416;
30 VAT=0.15;
40 CM=2.54;
```

providing that you have started correctly with CONST and that you have used the semicolons correctly. Fig 2.2 shows a simple program which makes use of the constant VAT to mean 0.15, as we would use it in a VAT program. Constants can be names and messages as well as numbers, as we'll see later. The constants should all be declared in a section that immediately follows the program title.

```
10  PROGRAM P2N5;
20  CONST VAT=0.15;
30  BEGIN;
40  WRITE('VAT ON $100 IS ');
50  WRITELN (VAT*100);
70  END.
```

Figure 2.2 A simple program that uses a constant, in this case a number constant. The **CONST** definition should come early in the 'header', just following the **PROGRAM** part.

The line which 'declares the constant' of VAT is situated between the program name and the BEGIN line. This is important, because a program cannot begin until any identifying name in it has been assigned with a value. In the program, the part which lies between BEGIN and END., we will use VAT as meaning the number 0.15. This meaning *must* be declared before the program begins. This way, the compiler has allocated memory space for the constant and is ready to use it before the program needs it. As we have seen, you might have to allocate several constants like this before a program starts. These constants need not all be 'real' numbers like 0.15 or 2.54. They could be integers such as 5 or 3000. They can also be letters, like 'Press any key', and this use of a constant replaces a lot of the purposes for which we use string variables in BASIC. This is important, because Pascal does not have string variables in the form that we use in BASIC. In the example, you can see the words 'VAT ON $100 IS' used as a phrase following WRITE, and the number constants 100 and VAT, of which VAT is a constant that has been assigned.

There's just one snag. When you run the program of Figure 2.2, you will find that the result is the line:

```
VAT ON $100 IS 1.50000E+01
```

This number is in 'standard form', which is nothing to do with Pascal, just a well-known way of writing numbers. If it isn't well-known to you, take a look at the Appendix to this book. If this still has no appeal, you can force any write action to produce numbers in the form that you want. What you have to specify is the total number of digits and the number that will follow the decimal point. For example, if you want the result of VAT to contain 7 characters, with no more than two following the decimal point, then you add the 'specifier' :7:2 to the number that you want to print. In this example, it would make line 40 read as

```
40 WRITELN(VAT*1000:7:2)
```

and the result would then be printed on the screen with only two places of decimals, as 15.00. This is a much more readable form. Note that the 7 refers to characters, including the decimal point itself. If you specify fewer characters than will be needed, the output will still be in the correct form, and there will always be a space ahead of the number unless a negative sign is present. If you are in doubt, then, you can use this 'formatting' statement to give the correct

number of decimal places, and use a 1 for the length number. Using :1:2 would therefore give you two decimal places, but you could still print numbers like 1256.45. The difference is that if you use :7:2, then seven spaces will be reserved for the number whether it needs seven or not. This can be useful when you are printing numbers in columns.

VARIABLES

The next obvious step is to see how we can incorporate a variable into a program. Suppose that we extend this to a variable entered from the keyboard? The standard identifier words for reading an input are READ and READLN, which are used in very much the same way as WRITE and WRITELN. What we are going to type is a number which will have to be assigned to a variable name, another kind of identifier. This is very close to the use of INPUT in BASIC, except that we can't make use of READ to print a message. There's nothing that corresponds to the BASIC statement: INPUT "HOW MANY?";A for example. In addition, the variable name that we shall use will have to be declared before the program starts. The correct place to carry this out is immediately following the CONST step.

A name for a variable is 'declared' to the compiler in much the same way as we treated a constant. In this case, we shall be entering an amount which is an exact number of units, with no fractions. We can therefore make the variable an integer. This is declared in line 30 by having

```
30 VAR SUM:INTEGER;
```

which establishes that there will be a variable called SUM which will be an integer. In more formal language, the variable type is integers. 'Integer' has almost the same meaning here as in BASIC, a whole number which lies between −32767 and +32767. BASIC allows an integer to go down to −32768, but that's the only difference. The fact that we have made SUM a variable of integer type means that only integer values can be assigned. The trouble is that you don't necessarily know this. Try the program, with an entry like 5.6. You'll find that the program does not reject this number, but the result that is printed is the result of VAT on the figure 5, not 5.6. This is an important point, because there is nothing to tell the user that the number which has been entered has been converted into an integer. It's up to you, the programmer, to do this.

The rest of the program in Figure 2.3 is straightforward. Note what the appearance of the output is, however. If you have used inputs of small numbers, there will be a large gap between the dollar sign and the amount. This is because the output has been specified as :7:2, meaning a total of seven characters with two following the decimal point. The program therefore allows spaces for four figures ahead of the decimal point—remember that the decimal point is included as one of the characters. If you want to reduce the space, use

```
10  PROGRAM P2N6;
20  CONST VAT=0.15;
30  VAR SUM:INTEGER;
40  BEGIN;
50  WRITELN('PLEASE ENTER AMOUNT');
60  READLN(SUM);
70  WRITE('VAT IS $');
80  WRITELN(VAT*SUM:7:2);
100  END.
```

Figure 2.3 A program which uses an integer variable as well as a constant. The type of variable and its name must both be declared in the header.

:1:2, which will still give you two decimal places, but with no excess spaces. There is always one space left in case of a negative sign. Note, in addition, another oddity, that there is no check on the size of an integer. You ought to be limited to a positive size of +32 767 for an integer, but if you enter numbers like 100 000 into the program of Figure 2.3, they will be accepted. The answer, however, is incorrect. What has happened is that the compiler accepts the number, converts it into the form that it uses for an integer (two-byte form), and then fits in what it can into the memory. If you add a line such as WRITELN (SUM); to your program, you will find that an entry of 100 000 is stored as the number −31 072, and 1000 000 is stored as 16 960. This avoids stopping the program because of an incorrect entry, but it also means that we have to test each entry rather carefully. It would be pleasant to think that I could pay a negative amount of VAT on £10 000 (and get it in dollars, too), but a program with errors like this wouldn't sell!

REALS

In contrast to an integer number, which can use only a limited range of whole numbers, all computers can handle numbers which are *real*. A 'real' number in this sense means the sort of number that we use in a lot of arithmetical calculations, numbers which can contain fractions, which can be positive or negative, and which can be larger than +32 767 or less than −32 767. As you have already seen, if you do nothing to the contrary, numbers like this will always be printed out in 'standard' form. A program input will also accept them in this form, or as a conventional decimal fraction. You can, for example, enter 1523.6 as 1523.6 or as 1.5236E3. A curious limitation, however, is that you cannot enter a number such as .25—it has to be entered as 0.25, with at least one digit ahead of the decimal point. You cannot impose a format on the input of a number, but once you have a number assigned to a variable name, you can do as you please with it.

A variable name which is used for a real number has to be declared before the program starts, as you might by now expect. This is still a variable declaration, so it has to start with the reserved word VAR, which is then followed by the name of the variable and its type. This time we shall be dealing with real numbers, so the name will be followed by :REAL so that the compiler can allocate the correct amount of memory. We can allocate both integer types and real types following VAR; we can in fact declare whatever variables we may want to use. The rules are much the same—there must be at least one space between VAR and the first variable name. If you have VAR on a line of its own, there must not be a semicolon following it. You can define a whole set of names as being of one type if you separate them with commas. You can, for example, use

```
90 SIZE,SCENE,ACT:INTEGER
```

to define three variables of type integer. There must be a colon between the names, which are identifiers, and the word INTEGER, which is also an identifier, but one of the identifiers which has already been assigned with a meaning. Words like CONST and VAR are used only once, with as many lines of identifiers as are needed for each one.

Figure 2.4 shows a program which this time assigns a real number. Line 30 declares that the identifier word CM is a variable of type REAL. The rest of the program is straightforward until line 70, in which both the message and the result are printed by one WRITELN statement. Notice how the two are separated. The first part of the message is contained between the single quotes, and a comma is used to separate this from the result CM/INCH:1:2. Note that this is a calculation, not just a variable name. Another comma follows, and then the last part of the message follows between another pair of single quotes. There has to be a space between the first single quote and the I of INCHES to prevent the last digit of the number being immediately next to the letter I. Once again, the formatting numbers :1:2 ensure two decimal places, with the number taking as much space as it needs. Notice that the comma does not have the spacing effect that it has in BASIC. In Pascal, the comma is used in a WRITE or WRITELN statement in the way that a semicolon is used in BASIC.

```
 10  PROGRAM P2N7;
 20  CONST INCH=2.54;
 30  VAR CM:REAL;
 40  BEGIN;
 50  WRITELN('PLEASE ENTER NUMBER OF CM.
');
 60  READLN(CM);
 70  WRITELN('THIS IS',CM/INCH:1:2,' INC
HES.');
 90  END.
```

Figure 2.4 A program which makes use of a 'real' number, meaning a number which is not an integer. Notice the use of the 'fielding' command in line 70.

Real numbers are used only for programs which involve calculation. Obviously, this would include a lot of financial and scientific work, but outside these needs, real numbers are surprisingly uncommon. The way that real numbers are stored in most computers means that the stored values are never precise, and the number which you see printed is always a 'rounded' version. This has nothing to do with whether you program in BASIC, Pascal or any other language, it's much more closely tied up with how the computer has been designed. If you want to make extensive use of real numbers in your programs, then you should be prepared to include steps which will round up or down when the number requires it. For example, you should round 3.999 99 to 4.0 and 27.000 01 to 27.0. This rounding is particularly important if you are dealing with number comparisons. If the computer is using the variable name of SUM to store a number, and this number happens to be stored as 197.999, then if your program is constructed so as to do something when SUM=198, nothing will ever happen. The computer does not accept that 197.999 is identical to 198, even if printing the two numbers on the screen produces identical results. You will know about this if you have programmed extensively in BASIC, but if you are learning Pascal in order to write some financial programs, you should know that problems like this are not caused by programming in Pascal!

OTHER VARIABLE TYPES

By the time any book on BASIC has reached this stage, the subject of string variables would have appeared. Now strings have a part to play in Pascal, as they have in any language, but the way that strings are handled is not quite so simple if you are making a transition from BASIC to Pascal. The reason is that STRING is not a pre-defined variable type; it isn't in the list of ready-made identifiers given earlier. Some versions of Pascal do include STRING (notably Pascal on the IBM PC); others do not—it's a matter of choice. Later, we'll look at how this identifier can be created but for the moment we'll look at the characters that make up a string.

Oxford Pascal, in common with all others, has a variable type called CHAR. CHAR means any character of the computer which is represented by an ASCII code. In Oxford Pascal for the Commodore 64, this includes the graphics characters as well as the ordinary alphabetical and digit characters. Now with this CHAR variable, we can do the actions that you associate with string variables in BASIC, and some more. Take a look at Figure 2.5, for example. Line 20 defines the variable LET of type CHAR. This means that the variable consists of one ASCII code, stored in one byte of memory. Line 60 shows how the number of the ASCII code can be obtained; it's ORD(LET), the equivalent of ASC(LET) in BASIC. To find the character whose code is known, you use CHR in Pascal as you would use CHR$ in BASIC. For example, CHR(NR) would give the character whose ASCII code was represented by the integer variable NR.

```
10  PROGRAM P2N8;
20  VAR LET:CHAR;
30  BEGIN;
40  WRITELN('PLEASE TYPE A LETTER');
50  READLN(LET);
60  WRITELN('THIS IS CODED',ORD(LET));
70  WRITELN('NEXT IS ',SUCC(LET));
80  WRITELN('PREVIOUS WAS ',PRED(LET));
100  END.
```

Figure 2.5 The **CHAR** type of variable, which means ASCII codes. There are BASIC counterparts for **CHR** and **ORD**, but not for **PRED** and **SUCC**.

Lines 70 and 80 show some new tricks, however. The identifier SUCC means 'successor', the next in line. If you typed F, then the next in line is G. If you typed 1, then the next is 2. This works strictly on ASCII codes, so the successor to " is # and so on. Line 80 illustrates the opposite action, using the identifier PRED. This means 'predecessor', the one that comes before in the ASCII list. The PRED of F is E, the PRED of 1 is 0, and the PRED of " is ! It's not quite so easily done in BASIC.

One last type of variable for the moment is Boolean. This has to be declared as type BOOLEAN, and will always have only two values, TRUE and FALSE. A Boolean variable is used where it's convenient to have just two values. You may want to check, for example, if a name has been found in a list, or if a particular number has been entered from the keyboard. Boolean variables have this deliberately limited range of values for just this purpose, to detect whether something is true or false. We'll be looking at the use of Boolean variables later, so we'll skip any examples for the moment. In the next chapter, we'll be looking at one of the glories of programming in Pascal, the use of 'free-range' variables!

3

TYPES AND PROCEDURES

FREE-RANGE VARIABLES

The 'standard' variable types of integer, real, char and Boolean, might, for a lot of programs, be all that you need. After all, BASIC provides you with integer, real and string variable types only. Pascal, however, allows you to define and use your own variable types. If you have only ever programmed in BASIC, this sort of liberation is difficult to appreciate, and to illustrate it fully needs quite a lot of thought as to how you would use it for your own purposes. Suppose, for example, that you wanted to write a program that dealt with coins. Your coins are the 1p, 2p, 5p and so on. This can be declared early in a program as

```
20 TYPE COIN=(ONE,TWO,FIVE,TEN,TWENTY,FIFTY,POUND);
```

but you can't use 1p, 2p and so on, because a name for new type that is declared with brackets like this cannot start with a digit. You can declare a TYPE which consists of digits, but only when the digits constitute a range, for example, 1..11 meaning any value between 1 and 11. Getting back to the example, though, in addition to variables of type integer, real, char and Boolean, you have type COIN. Type COIN can only take the values that you have defined, no others.

Now another trouble with this new-found freedom is that you probably expect too much from it. You cannot define a new type COIN, for example, to declare a variable of type CASH and have a statement WRITE(CASH). This is because WRITE (along with WRITELN, READ and READLN) will accept only the *standard* variables, such as integer, real, Boolean and char. The only exception to this is the 'string variable' which we shall be looking at later. The other thing is that though we can find the position of any item in the list by using ORD, we can't use CHR or anything similar to work in the opposite direction. For example, we can find that ORD(FIVE) is 2 (counting 0,1,2...), but we can't ask for CHR(2) and expect to get FIVE, because CHR operates on ASCII codes only. Take a look at Figure 3.1, which shows a few novelties. In this program,

```
 10  PROGRAM P3N1;
 20  TYPE COIN=(ONE,TWO,FIVE,TEN,TWENTY,
FIFTY,POUND);
 30  VAR CASH:COIN;
 40  BEGIN;
 50  PAGE;
 60  WRITELN('COINS');
 70  FOR CASH:=ONE TO POUND DO
 80  BEGIN;
 90  WRITELN('POSITION IS',ORD(CASH));
100   END;
120   END.
```

Figure 3.1 Using a variable type which is defined in the header. This must be done before a variable of this type can be declared. A **FOR** loop has also been used in this program and unlike the BASIC **FOR** loop, it doesn't need a **NEXT**.

type COIN is defined, and the variable CASH is of type COIN. The program then starts in line 40. PAGE; in line 50 has the effect of clearing the screen—if the output is to a printer, this gives a form-feed. The heading COINS is then printed, and a loop starts. Now this is a FOR type of loop, and because it does the same as a FOR...NEXT loop in BASIC, I have put it in without introduction. The loop starts with

```
70 FOR CASH:=ONE TO POUND DO
```

which includes the first assignment that you have seen. The main difference here is that two signs are used in an assignment. The equality sign by itself is always reserved to mean genuine equality, but the combination of colon and equality sign (:=) is used for assignment. You don't need to use LET, but you must remember that colon. The next surprise in this lot is that the loop can use the variable CASH, with its first and last values of ONE and POUND. This is because when the type COIN is defined, the positions of the words in the list are numbered 0 to 6. As far as the compiler is concerned, then, this is just a FOR 0 TO 6 type of loop. A variable like CASH is sometimes called an 'enumerated' type, meaning that each item can be coded as a number which is its position in the list.

We come to a very important difference at the end of line 70, however. The word DO must *not* be followed by a semicolon, and it has to be followed by lines that define what is to be done. These lines should start with BEGIN and end with END;. Not, you'll notice, END., but END; with a semicolon. Whatever you place between DO and END; is done as many times as the loop goes round. In this example, what we do is to print ORD(CASH), meaning the coded order number of each item. After the end of the loop, the whole program ends, this time using END. In a lot of Pascal printouts, you'll see these loops indented so

as to make it more obvious where the loops start and finish. This is something we can do later, but for now we'll concentrate on what's happening.

The use of these 'free-range' variable types extends beyond the descriptive list that we have looked at. You can, for example, define a type which is part of a range of a standard type, like integers, or a newly defined type like COINS. This type of variable is referred to as a 'subrange', because it is part of a wider range. You can, for example, define a type SINGLEDIGIT which consists of the digits 0 to 9. This would be programmed as

```
30 TYPE SINGLEDIGIT=0..9;
```

and there must be *two* full stops between the numbers. This is important—you will get error messages if you use any other number of dots. Remember that you can get error messages where you might not expect them, and you always have to look carefully at lines which come before the one in which an error is reported.

Take a look now at an example of this in action in Figure 3.2. This defines in line 20 a type SPOTS which consists of the numbers 1 to 6. Now these are taken as integers, and so the type SPOTS is really just another type of integer. When

```
10  PROGRAM P3N2;
20  TYPE SPOTS=1..6;
30  VAR DOMINO:SPOTS;
40  BEGIN;
50  WRITELN('HOW MANY SPOTS?');
60  READLN(DOMINO);
80  END.
```

Figure 3.2 Another defined type. In this case, however, **SPOTS** consists of an existing type, a selection of integers. Line 60 will reject any entry which is not of the correct range.

we define the variable DOMINO, then, as being of type SPOTS, this is another way of saying that any number which is assigned to DOMINO is an integer number which has to be between 1 and 6. We can use integers in READ and WRITE statements, unlike some other types, and so line 60 can read a number from the keyboard and assign it to the variable DOMINO. There's an interesting difference, however. If DOMINO were just another integer variable, we could read any number in, and values between −32 767 and +32 767 would be stored. When we have defined this limited range, though, anything outside this range is an error. If, when the program reaches line 60, you enter a whole number which is somewhere between 1 and 6, all is well. You can even enter numbers like 4.65 or even 6.75, because these are chopped to the integers 4 and 6 respectively. If you try to enter 0 or 7, or anything higher, however, you will get the error message

```
VARIABLE OUT OF RANGE, LINE 60
```

and the program will stop. This isn't always very desirable, but if your program is one in which you want an incorrect entry to be caught in this way, this is how it is done. We'll look at more variable types and subranges as we go on, but for the moment, we have to start looking at more of how we make use of variables.

PROCEDURES

We have seen already that a program consists of a heading, definitions, a BEGIN statement, action statements, and then an END. In the main program, between the BEGIN and the END, you place all the actions, in sequence, that the program will carry out. Now in short programs these actions will be simple ones, and they can be written between the BEGIN and the END of the main program. As your programs become longer, however, you will need to break them into sections, if only for the sake of planning. Just as you can break a BASIC program into a core and a set of subroutines, you can break a Pascal program into a main block and a number of procedures. The similarity between the languages ends there, though. A procedure is called into action by using a name, its identifier. In addition, values can be passed to a procedure in ways that are not used by subroutines. The use of a procedure is therefore something that needs rather more thought than the use of a subroutine. You can look on a Pascal program as one large procedure which has minor procedures nested inside it, and you should try to design Pascal programs with this idea in mind.

Take a look now at the program in Figure 3.3. This starts in the usual way, and defines the variables FIRST and SECOND as real numbers. A PROCEDURE is then defined in lines 30 to 60. This procedure is called ROOTSUMSQUARE, and following the identifying name of the procedure, we have to put in the numbers that this will use. The important thing about a PROCEDURE is that these variable names do not have to be the same as the ones that we use in the main program, or anywhere else. In the definition, ROOTSUMSQUARE uses numbers A and B which are real numbers. This is put into the definition of the procedure by using (A,B:REAL). This part is the 'header' of the procedure, and now follows the action. This is in line 50, and if the sight of all these brackets is disturbing, then aim for what happens in the middle. Here, the quantity SQR(A) is added to SQR(B). In Oxford Pascal, SQR means 'the square of', and the effect of this will therefore be to add the square of A to the square of B. In the outer brackets, SQRT is used to take the square root of the amount that has been calculated from SQR(A)+SQR(B). The formatting numbers :1:3 will cause the answer to be printed with three places of decimals. Since all of this is placed following a WRITELN identifier, the result will be printed in this form. Finally, the word END; marks the end of this section, but not of the program as a whole.

Now the main program starts. You are asked for two numbers, which are entered as variables FIRST and SECOND. Line 110 then calls the procedure.

This is done by using the name, and placing the two variable names within brackets. It doesn't matter that these variable names are not the same as the A and B that were used in the definition, the important thing is that they are of the same type. In this example, this type is real numbers. When the program reaches line 100, you are asked to enter two numbers. You can do this by typing

```
10  PROGRAM P3N3;
20  VAR FIRST,SECOND:REAL;
30  PROCEDURE ROOTSUMSQUARE(A,B:REAL);
40  BEGIN;
50    WRITELN(SQRT(SQR(A)+SQR(B)):1:3);
60  END;
70  (*MAIN PROGRAM STARTS HERE*)
80  BEGIN;
90  WRITELN('TWO NUMBERS, PLEASE');
100  READLN(FIRST,SECOND);
110  ROOTSUMSQUARE(FIRST,SECOND)
130  END.
```

Figure 3.3 A **PROCEDURE** in use. The procedure must be defined before the start of the main program that calls it. The procedure is activated by using its name, along with any values that have to be used in the procedure.

one number, pressing RETURN, and then typing the second number and pressing RETURN. A much neater method is to type the first number, then a space, then the second number, and then press RETURN. You can *not*, however, use the method that BASIC has of typing both numbers separated by a comma, and then pressing RETURN. When the numbers are entered, the procedure then prints the value of the root of the sum of the squares.

The variables A and B exist only within the procedure. If you try to print their values near the end of the program (a line 115, perhaps), you will find that you get an error message about UNDECLARED IDENTIFIER. This means that A and B were not declared as variables at the start of the program. The fact that they were declared as reals in the procedure refers to the procedure only, and you can print the values of A and B while the procedure is running. If, for example, you add a line 45 which prints values of A and B, you will see the values appear which you have entered. The significance of this is that you have entered values for variables FIRST and SECOND. These values are then temporarily transferred to variables A and B for the procedure. This is completely automatic, you do not have to use instructions such as A:=FIRST and B:=SECOND. This is very different from the methods that you have to use in Commodore 64 BASIC, though the BASIC of the BBC Micro contains both procedures and this method of passing values. We'll look later at the principle of passing values back from a procedure.

PLANNING

One of the most important points about procedures is how they affect planning of programs. How, for example, do we plan the simple illustration of Figure 3.3? Figure 3.4 illustrates a version of one method which is favoured by many programmers. The left-hand side shows the steps of the main program, with the

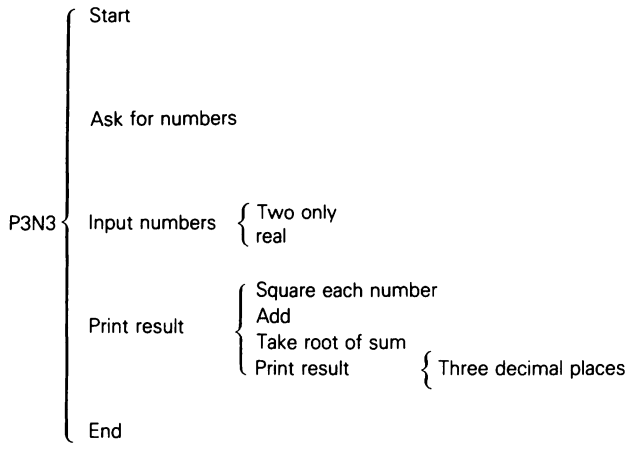

Figure 3.4 A simplified version of a popular planning method. The main steps are shown down the left-hand side, and detail to the right. Each curly bracket indicated a further stage of detail of a step.

main action shown as one step. The curly brackets are then used to show where more details are needed. This has been used in the example to show variable names, and also to show what has to be done in the ROOTSUMSQUARE procedure. The important point is that a procedure is designed in very much the same way as the main program is designed. You can design a procedure without constantly having to refer to the main program, because the variables that are used within a procedure need not bear the same names as those used in the main program; the only essential feature is that they should be of the same type. In general, if you define variables for the main program at the start of the program, these variables can also be used in the procedure. If you define variables inside a procedure, these variables are 'local', they are used only in the procedure, and simply don't exist when the procedure is not running. This is very different from a subroutine, which can use or define variables which will from then onwards be available in the main program.

MORE PROCEDURE PLANNING

Let's look now at a short program, starting with the design steps. This is to be a program which will convert a list of Celsius temperatures into their Fahrenheit

equivalents. The planning, such as it is, is shown in Figure 3.5. On the left-hand side, the main steps of the program are listed as header, variables, range, convert and print. The only detail which is included in this list is the range of temperatures, 0 to 100 in steps of 10 Celsius degrees. The curly brackets now open into more detail. The variables will be called F and C and will be real numbers. We could have chosen integers here, but if you alter the range you will need to use reals for the Fahrenheit quantities at least. The range and steps will be dealt with by using a loop, and the conversion will be done by a procedure. The main feature of the procedure is the use of the conversion formula.

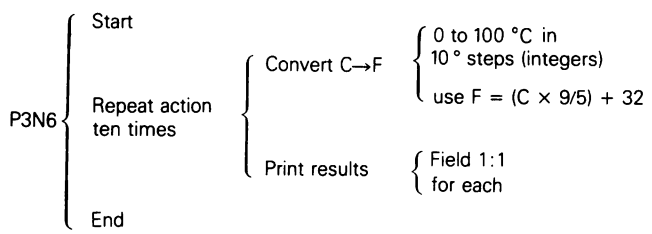

Figure 3.5 The planning for the temperature conversion program.

Now this program illustrates two important points which never arise in Commodore 64 BASIC. One is that a FOR loop in Pascal does not have the STEP command. In BASIC, you could obtain steps of ten degrees by programming:

```
FOR N=0 TO 100 STEP 10
```

but in Pascal we cannot do this. The only FOR loops that we can use, in fact, are the incrementing loop, which we have used already, and the decrementing loop, which is programmed by something like

```
for N:=10 DOWNTO 0 DO
```

—but only a step of −1 is permitted here. If we want to make a step, we have to include it in another part of the loop, and this will have to be done in this example. The other point which this program illustrates is one way in which a value can be passed back from a procedure. This is not quite as straightforward as you might think, because any quantity that is to be passed to the procedure has to be defined in the procedure. If it is defined in the procedure, however, its value is lost when the procedure ends! We need a way around this, and in the following two examples, we shall show how we can make use of 'global' variables, which are defined in the main header and used in all parts of the program.

Figure 3.6 shows the program which has been drawn up from the plan. As always, looking at a finished program gives you no idea of what the steps were in writing it, so we'll look at the program in the order in which it was written. When you write a Pascal program from a plan, never allocate any line numbers. Leave the allocation of line numbers to the Commodore 64 when you enter the program. This avoids continual re-numbering when you have to squeeze yet another definition into the start of the program! The main program in lines 90

```
10  PROGRAM P3N6;
20  VAR F,C:REAL;
30  N:INTEGER;
40  PROCEDURE CONVERT(CIN:REAL);
50    BEGIN;
60    F:=(CIN*9/5)+32;
70    END;
80  (*MAIN PROGRAM*)
90  BEGIN;
100  FOR N:=0 TO 10 DO
110    BEGIN;
120    C:=N*10;
130    CONVERT(C);
140    WRITELN(C:1:1,' CELSIUS IS',F:1:
1,' FAHRENHEIT');
150    END;
170  END.
```

Figure 3.6 The conversion program. The main program is in lines 100 to 170. This calls the procedure **CONVERT** which is defined in lines 40 to 70. The variable **CIN** is 'local' to the procedure.

to 150 was written first. This starts in the usual way with BEGIN, and then the loop starts in line 100. This loop, extending from line 100 to line 150, carries out the instructions of lines 110 to 150, just as if this set of lines constituted one single statement. A set like this is often called a 'compound statement'. In the loop, the loop variable N is multiplied by 10 and assigned to C. This is the equivalent to using a STEP of 10 in BASIC. The instruction CONVERT(C) is the call to the procedure, and C represents the Celsius temperature. The procedure will convert this to Fahrenheit, using variable name F. Line 140 then prints both amounts, using one decimal place if needed. With the range of quantities that we have chosen, there will never be a figure in the decimal position.

We could have started with the header, because we have planned the use of variables. It's usually a good idea to write the header later, however, just in case you have changed your mind, or thought of something that you want to incorporate. If you need another variable, then, it's easy to put it into a revised plan, and then write the header. This includes the program name and the variable definitions. With that done, only the procedure remains to be written. This is where some care is needed. The name of the procedure is CONVERT, but you have to watch what goes in the brackets. If you use C, you will have to define it as a real, because the compiler will not allow you to use an undeclared variable here. It doesn't matter that the variable was declared in the main section, this doesn't count! If you use C here, then, it must go in as C:REAL,

and any values which it gets will be lost when the procedure ends. This is why the line C:=N*10 has been used in the main program. To avoid confusion, then, a quantity CIN has been used in the procedure. The use of the formula in line 60 makes use of variable F. Now this wasn't defined in the procedure name, but it *was* defined in the main program header. The difference is that F is not a quantity that is passed to the procedure. It is a 'global' variable, one that is used both by the main program and by the procedure. This global variable will be assigned with a value in the course of the procedure, and this value can also be used by the main program, in line 140.

We could, in fact, simplify the program further by not passing any values directly. Figure 3.7 shows the program re-written so that there are no brackets following CONVERT and with the variable C used. This works also, because

```
10  PROGRAM P3N7;
20  VAR F,C:REAL;
30  N:INTEGER;
40  PROCEDURE CONVERT;
50    BEGIN;
60    F:=(C*9/5)+32;
70    END;
80  (*MAIN PROGRAM*)
90  BEGIN;
100  FOR N:=0 TO 10 DO
110    BEGIN;
120    C:=N*10;
130    CONVERT;
140    WRITELN(C:1:1,' CELSIUS IS',F:1:
1,' FAHRENHEIT');
150    END;
170  END.
```

Figure 3.7 A slightly amended conversion program. This time, **C** is a 'global' variable which is passed to the procedure to be converted.

both C and F are being used as global variables. The important point to remember here, then, is that any variable name which is used within brackets in a procedure definition will be local, it will not have a value outside the procedure. You can, in fact, use the same variable name for an entirely different purpose outside the procedure. You can regard the procedure as a mini-program which can make use of any variables that exist in the main program, but which keeps its own variables to itself. Much later, we'll see how a variable that is defined within a procedure header can be passed to the main program and its value used.

FUNCTIONS

Defined functions are a feature of Commodore 64 BASIC, and I'll assume that you have probably made use of them. When programs make use of numbers, as ours have so far, the difference between a defined function and a procedure is not obvious. The main difference is that a function must return at least one number to the main program. A procedure, by contrast, might return nothing or can return several variable values, which can be numbers or other types. Another important difference is that a function action is purely a number action. A procedure can include lines that print something; a function cannot do this. A function must end with an assignment, and it's the value which is assigned in this last step of the function that will be passed to the main program.

Because a Pascal defined function is not written in the same way as a BASIC defined function, we need a couple of illustrations. Suppose we design a program which reads in diameter values and calculates the area of circles. The important part of this will be a function which will accept a real variable, the diameter, and return another real, the area. Figure 3.8 shows the result, skipping the planning stages. The function portion is the one that you should examine in detail. The function has to be named, and the name that has been allocated is AREA. This will have a value passed to it, and this has to be defined

```
10  PROGRAM P3N8;
20  CONST PI=3.1416;
30  VAR D:REAL;
40  FUNCTION AREA (DIAM:REAL):REAL;
50  BEGIN;
60   AREA:=PI*SQR(DIAM/2);
70  END;
80  (*MAIN PROGRAM*)
90  BEGIN;
100  PAGE;
110  WRITELN('PLEASE ENTER DIAMETER');
120  READLN(D);
130  WRITELN('AREA IS',AREA(D):1:2);
150  END.
```

Figure 3.8 Illustrating a defined function. This is very similar to the Commodore 64 BASIC defined function and in this example it calculates the area of a circle.

within the brackets. This portion is the header for the function. The chosen variable for diameter in the function is DIAM, and it will be a real variable. When the brackets close again, though, there is another REAL declaration. This is for the name of the function, AREA. We are going to use the word AREA for two purposes, one as the name of the function, and one as a variable name for a quantity. This is the reason for the :REAL; which follows the brackets.

The action of the function now has to be placed between the words BEGIN; and END;. This consists of an assignment to the variable name AREA, and in this example, it's just the equation for the area of a circle, pi times the square of half of the diameter. The constant PI is a global one which has been declared in the main header. You can equally well put the definition of PI into the function, but this just gives the function more work to do. If, as is likely, this main program consisted of a loop in which several values of diameter would be entered, then the logical place for the assignment of PI is in the main program. This makes the assignment once each time the program is used, rather than each time the function is called.

The other main point now is how the function is used, and this is straightforward. The name of the function is also the variable name for its result, and this can therefore be printed or assigned in the normal way. What you have to remember, though, is that there has to be a value passed to the function, and this value is represented by a variable name which is enclosed in brackets following the name of the function. The same rules apply here as applied for procedures. You can pass more than one variable name to the function, provided that the function has been prepared for them in its header. You can use a global variable if you like, and in this case, using a global variable for D would make the program look as in Figure 3.9. In this version, there are no brackets following AREA because no local variables are being declared. The variable D exists in the main program as a global variable, and is used also in the function.

```
10  PROGRAM P3N9;
20  CONST PI=3.1416;
30  VAR D:REAL;
40  FUNCTION AREA:REAL;
50  BEGIN;
60   AREA:=PI*SQR(D/2);
70  END;
80  (*MAIN PROGRAM*)
90  BEGIN;
100  PAGE;
110  WRITELN('PLEASE ENTER DIAMETER');
120  READLN(D);
130  WRITELN('AREA IS',AREA:1:2);
150  END.
```

Figure 3.9 Using a global variable in the area program. This avoids having to declare **D** in the header, but it means that you *must* use variable **D** when you call the function.

PASSING SEVERAL VALUES

Suppose we need to pass several values to a function? The definition of the function only has to be amended if you want to use local variables, as the

previous examples have indicated. Since the use of local variables is the most likely method that you will use, however, I'll illustrate it. Using local variables is useful because it lets you design the function without having to worry too much about what your variables are called in the main program. In the next example, then, we'll calculate the volume of a cube from the dimensions of three sides. I'm sorry to keep harping on with number examples, but until we look at how strings are handled in Oxford Pascal, there isn't much choice. See Figure 3.10.

```
 10  PROGRAM P3N10;
 20  VAR LEN,WID,HT:REAL;
 30  FUNCTION VOL(L,W,D:REAL):REAL;
 40   BEGIN;
 50   VOL:=L*W*D;
 60   END;
 70  (*MAIN PROGRAM*)
 80  BEGIN;
 90  WRITELN('PLEASE ENTER THE FOLLOWING
:');
 100  WRITE('LENGTH ');
 110  READLN(LEN);
 120  WRITE('WIDTH ');
 130  READLN(WID);
 140  WRITE('HEIGHT ');
 150  READLN(HT);
 160  WRITELN;
 170  WRITELN('VOLUME IS',VOL(LEN,WID,HT
):1:2);
 190  END.
```

Figure 3.10 Passing three variable values to a function. The important point to note is how the three are defined, and that the names in the function need not be the same as the names in the main program.

We start by designing the function, and we'll use the local variables L, W and H to mean the length, width and height of the cube. This allows us to use any other variable names that we like in the main program. In the action of the function, these three will be multiplied together to give the quantity VOL. As usual, we design the main part of the program first. This will consist of a set of prompts, with a READLN input for each prompt. Once the last item has been entered at the keyboard, the variables LEN, WID and HT will have been allocated. To get the volume printed, then, we have to instruct the program to print the value of VOL(LEN,WID,HT). This will call the function, and pass these

three values to it. Since we have followed this with :1:2, the result will be printed with a maximum of one decimal place.

When we work on the header, the variables are LEN, WID and HT, and these have to be declared as reals. The function must then be defined. It will use variables L,W and D, and will calculate the volume by multiplying these three together. The result, VOL, is also real. Passing three variables, then, is no more difficult than passing one. The main item to remember is that the order of variables must be maintained. When the three quantities are simply multiplied together, as they are in this case, the order is not really important. When, however, different things will be done with each quantity (like raising the first quantity to the power of the second and adding the third), then order *is* very important. The order of the variables in the function definition must be the same as the order in the calling expression. If you define FUNCTION DOIT (X,Y,Z:REAL):REAL; and then call with DOIT(A,C,B), in which A equates to X, B to Y and C to Z, then you will have trouble on your hands unless the function is symmetrical. Symmetrical, in this sense, means that the quantities are treated in the same way, all added or multiplied for example.

4

BUILT-UP AREAS

ARRAYS

In BASIC, you have simple variables, such as integers, reals and strings; and you also have one structured variable, the array. By 'structured', I mean that an array name like A means not just a single value, but a set of values which carry distinguishing names like A(1), A(2) and so on. Pascal is very well equipped with 'structured' variables, or structured data types, as they are called. One of these types is the array and for a number of reasons, we need now to look at what it is and what we can achieve with it. As you might expect, an array has to be declared at the start of a program and this declaration will include a name (identifier), the number of elements in the array, the type of data that is to be stored and some other points that we'll leave for later. This is just what you would expect from our experience of arrays in BASIC. When you use a DIM statement, such as DIM AB$(20), in BASIC, you are specifying the name AB, the type (string) and the number of elements (from 0 to 20, a total of 21). The main difference in Pascal then will be the way in which an array is defined rather than the information which is used.

Suppose, for example, that we want an array called CLASSMARKS to hold a set of 20 integer numbers. The declaration that we need for this looks something like this:

```
30 VAR CLASSMARKS:ARRAY[1..20] OF INTEGER;
```

This provides the name of the array, which is CLASSMARKS, the number of items (1 to 20) and the fact that each item will be an integer. This variable declaration can be made along with other declarations following a VAR in the header (which could be on a previous line). The important point to note here is the use of the *square* brackets. If you forget that you are writing Pascal and not BASIC, it's easy to refer to an item as CLASSMARKS(12), when you should be using CLASSMARKS[12] instead. The error mesage that you get when you

do something like this will not necessarily remind you of what has gone wrong. Since the array in this example is an array of integers, you can use READLN to get each item of the array.

Figure 4.1 shows an example of this array in use, and also some more formatting. In this program, a set of twenty marks is obtained. There's nothing to stop you from entering numbers like 5000, but the program assumes that the items will be between 0 and 99. Later on, we'll see that this can be checked. The items are entered in a loop, using principles that should be familiar by now. When all of the items have been entered, the screen clears, and the title CLASSMARKS is printed centred. This is done by typing a 'field size' number following the word CLASSMARKS. The field size number represents the total size of string that is printed, and if the word is less than this size, it is padded with blanks. By using a positive number, we force the word to be printed with any of these padding blanks on the left-hand side. The choice of the number 25 with CLASSMARKS (which has ten letters) means that 15 spaces will be printed to the left of the name. This can be used to centre any name as follows

1. Count the characters in the name, for example, 15. Take half this number, and round to a whole number. In this example, make it 7.
2. Add 20, and use the result as a field number. In this example, you would use :27 as the field. The title can be a string constant, but more care is needed to centre a variable (illustrated in Chapter 8).

If, incidentally, a negative 'field' number is used, the excess spaces are printed

```
10  PROGRAM P4N1;
20  VAR  CLASSMARKS:ARRAY[1..20] OF INTEGER;
30  NUM:INTEGER;
40  BEGIN
50   FOR NUM:=1 TO 20 DO
60     BEGIN;
70     WRITE('MARK',NUM,' - ');
80     READLN(CLASSMARKS[NUM]);
90     END;
100   PAGE;
110   WRITELN('CLASSMARKS':25);
120   FOR NUM:=1 TO 20 DO
130     BEGIN;
140     WRITELN('ITEM ',NUM:2,' GOT ',
        CLASSMARKS[NUM]:2,' MARKS');
150     END;
170 END.
```

Figure 4.1 Illustrating an array of integers which is entered in one loop and printed in another.

to the *right* of the name. This can be useful for spacing the next name, but is not so useful for a heading.

The items of the array are then printed out in order, using the variable name NUM as the item number and also as the array number. The printing line, line 140, uses spaces following names, and field numbers of 2 following numbers to space out the items in a reasonably pleasing way. If you have been used to the action of TAB in BASIC, the use of the field numbers can often be very difficult to adapt to. The rule is to use a positive field number when you want to start a word away from the left-hand side of the screen, and a negative number to space the next item along so that it will fit neatly. This technique is illustrated in Figure 4.2, which is a re-write of the CLASSMARKS program. It also incorporates a safeguard against entering the wrong items for the marks.

```
 10  PROGRAM P4N2;
 20  TYPE MARKS=0..99;
 30  VAR  CLASSMARKS:ARRAY[1..20] OF MAR
KS;
 40  NUM:INTEGER;
 50  BEGIN
 60   FOR NUM:=1 TO 20 DO
 70     BEGIN;
 80     WRITE('MARK':10,NUM:3,' - ');
 90     READLN(CLASSMARKS[NUM]);
 100     END;
 110   PAGE;
 120   WRITELN('CLASSMARKS':25);
 130   WRITELN;
 140   FOR NUM:=1 TO 20 DO
 150     BEGIN;
 160     WRITELN('ITEM':-5,NUM:4,'GOT':5
,CLASSMARKS[NUM]:4,'MARKS':6);
 170     END;
 190  END.
```

Figure 4.2 The **CLASSMARKS** program rewritten to illustrate how field numbers can make the appearance of the output neater.

In detail, the program defines TYPE MARKS as being the numbers 0 to 99. This means that any number which does not fall within this range will be rejected. The rejection takes the form of stopping the program if a faulty mark is entered. This means that you would have to start all over again, and we'll look at ways round this later. Though it's not stated, these numbers are integers. The variable CLASSMARKS is now defined as an array of type

MARKS, so that the items of the array must also be integers in the range 0 to 99. The other changes concern the entry and output writing lines. In line 80, the word MARK will be printed on the screen in a field of 10. Since the word uses only four letters, it will have six spaces added on the left-hand side, equivalent to a tabulation of six spaces in. The variable NUM has a field of 3, so a single digit number will have two spaces added and a two digit number will have one space added, once again on the left-hand side. This makes the entry portion look much neater than in the original version. In the output section, a blank line has been left under the title. The printout in line 160 has also been altered to make use of field sizes. The word ITEM has been printed set to the left, with one space following it. The first number, NUM has a field size of 4—but there is a curiosity here because the actual field appears always to be one space less. As long as it's consistent, we don't have to worry too much. The word GOT uses a field size of 5, so that there will be two spaces ahead of the word (between NUM and GOT), and the array number has a field of four. Finally, the word MARKS uses a field size of 6 to put one space between the preceding number and the first letter of the word. The net result of all this work is to make the display considerably neater. What we need now is a way of incorporating string variables as well as string constants into our programs. Another point concerns checking of data. Because this program has defined CLASSMARKS as being an array of MARKS, and MARKS is a type which means the integers from 0 to 99, you can't enter a number like −10 or 120. These will be rejected with a VARIABLE OUT OF RANGE, LINE 90 error message. This is not always a desirable method of checking, however. If, for example, you have entered 99 items in a list of 100, and you make this kind of mistake on the last item, the error message tells you that you have lost everything, and you will have to start the program again. This is a compiled program, remember, so you can't just use the old BASIC trick of commanding a GOTO to get you back into the program. This method of protection, then, is suitable only when it doesn't matter if the program might stop when an incorrect item is entered. If you want to be able to check an item without stopping the program, then an IF test (like BASIC) is a preferable method. There's no reason why the result should not then be assigned to an array which has restricted values. The golden rule is that your program should never bomb out when the unlucky user (you, perhaps) has just entered a lot of data.

STRINGS AT LAST

A string in Oxford Pascal can be regarded as an array. This is true in other versions of Pascal, but in many of these there is a ready-made identifier STRING as one of the main variable types. The Pascal of the IBM PC, for example, allows you to use two identifiers, STRING and LSTRING. The main difference between this and Oxford Pascal is that we do not have these string

variables ready-made, we have to define them for ourselves. A string is an array of characters, and we always define it as a special type of array, the 'packed' array. Packing has rather more meaning for other data types, but the principle is that a lot of data items take up more memory space than they need to when they are in array form. Packing means that every unused space is eliminated, allowing the data to take up the minimum possible amount of memory. Packing is not particularly desirable if you need to be able to get at each item of the array, and for some types of data you would not want to use packed data. In resident mode, no unpacking instructions are available for digging single items out of packed arrays, so it's wise to avoid packing for a lot of array uses. Whether an array is packed or not, however, you can always assign one array to another variable name. For example if you have variables LIST and COLLECTION, both of which are array types, you can use LIST:=COLLECTION to make a copy of the array COLLECTION into the name LIST. In BASIC this could be done only by using a loop such as:

```
1000 FOR N= 1 to 100
1010 LIST(N)=COLLECTION(N)
1020 NEXT
```

Let's get back to the strings, however. Figure 4.3 shows a very simple string variable reading program. The word REPLY is defined as a packed array of 20 items of type CHAR. This means that REPLY will always consist of 20 ASCII codes. The number can't be ignored. It means that there will always be 20 characters in REPLY. If you enter only two characters, the rest will be filled with spaces. If you enter 22 characters, the last two will be lost. A packed array of characters is not such a useful and flexible thing as a string in BASIC. You can

```
 10  PROGRAM P4N3;
 20  VAR REPLY:PACKED ARRAY[1..20]OF CHA
R;
 30  BEGIN;
 40  PAGE;
 50  WRITELN('YOUR NAME':20);
 60  WRITELN;
 70  WRITELN('PLEASE TYPE YOUR NAME');
 80  READLN(REPLY);
 90  WRITELN('O.K. ',REPLY,' LET''S GO')
;
 100   WRITELN(REPLY:2);
 110   WRITELN(REPLY:-2);
 130   END.
```

Figure 4.3 A program which reads your name as an array of type **CHAR**, and prints it with different fieldings.

obtain an action which is rather like that of LEFT$ in BASIC by using a field width which is smaller than the string, as line 100 shows. The opposite action, that of RIGHT$ in BASIC, is not achieved by using a negative field length, however, as the program demonstrates. You might expect that this was because of the spaces that are used to fill up the string, but even with a twenty-character name, you don't get the expected RIGHT$ action. Notice, too, that you can put an apostrophe mark into a string by typing *two* apostrophes! Another point that is worth mentioning is that if you forget to specify a PACKED array, you will get error messages concerning the READLN and WRITELN actions on the string. These instructions will work on individual numbers or characters and on packed arrays of characters, but not on any other data type.

In Oxford Pascal, string actions look rather complicated because assignment is not quite so easy as you might think from the manual. If, for example, you have defined ANSWER as a PACKED ARRAY [1..4] OF CHAR, then you cannot make the assignment

```
ANSWER:='HOW'
```

because HOW contains three characters and the string must contain four. Assignment will not put the extra space in for you. On versions of Pascal which have a pre-assigned LSTRING type, this problem does not arise. It's really only a problem, however, if you are 'thinking in BASIC'. You can assign a string *constant*, such as DS='STRING CONSTANT', in a lot of places where in BASIC you would use a string variable. When you really need to use a string variable is when you are inputting or outputting strings, and the packed array of CHAR permits this with no problems. The thing that you really have to be careful about is any attempt to assign a string, because only a string of exactly the same structure is compatible. If your strings are packed arrays of 20 characters, then only another packed array of 20 characters can be assigned. This is very difficult to get used to when you have been accustomed to the free and easy ways that BASIC has with strings.

What do you do, then, when you *must* assign one string to another name? The answer is that if you have designed your program in such a way that you have to do this with strings of unequal length, you haven't designed the program correctly. If you are completely stuck, though, you *can* assign a short string to a string name that will take a longer string. The trick is to assign letter by letter and then fill with spaces. If you can be sure that the string that you have assigned to is empty, you don't even need to carry out the filling action. Since a string is really an array of ASCII characters, you can refer to each letter in the string by its place number. For example, if you have the name STRING being used for a packed array of CHAR, then STRING[1] is the first letter, STRING[2] is the second letter and so on. You can therefore make assignments either to or from these elements. For example, if you have the instruction

```
WORD[6]:=STRING[6]
```

then the number 6 character in the string WORD is now the same as the number

```
 10  PROGRAM P4N4;
 20  TYPE ANSWERS=PACKED ARRAY[1..8]OF C
HAR;
 30  NOTES=PACKED ARRAY[1..20]OF CHAR;
 40  VAR DAYS:ANSWERS;
 50  DAYNOTES:NOTES;
 60  N:INTEGER;
 70  BEGIN;
 80  DAYS:='MONDAY  ';
 90  FOR N:=1 TO 8 DO
100     BEGIN
110     DAYNOTES[N]:=DAYS[N];
120     END;
130  FOR N:=9 TO 20 DO
140     BEGIN;
150     DAYNOTES[N]:=' ';
160     END;
170  WRITELN(DAYNOTES);
190  END.
```

Figure 4.4 How to transfer characters from one array to another. You can't simply equate the array names because arrays of different sizes are not compatible.

6 character in the string STRING. Figure 4.5 shows how a complete string can be assigned in this way. Two types have been declared, one as a packed array of [1..8], the other as another packed array of [1..20]. In line 80, the word MONDAY is assigned to the variable DAYS, which is of type ANSWERS, an array with eight characters. In this case, two of the characters are spaces. We can now assign this word, character by character, to the name DAYNOTES. This is done in the loop which starts in line 90. When eight characters have been shifted, the rest of DAYNOTES should be filled up with spaces. Notice I said *should*, not *must*. You can, if you like, skip the filling with spaces, but you may get odd things appearing if you do. If you fill with spaces, you can be sure that when you print the new string, it will be exactly as assigned. If you don't fill with spaces, you are likely to get odd letters and other rubbish appearing. This is because you have no way of ensuring that the memory which has been used does not already have something stored in it.

The use of STRING[N], where N is an integer, is equivalent to the way that we use MID$(S$,N,1) in BASIC. In general, working with strings character by character in Oxford Pascal is easy, and anyone who started with one of the ZX computers will adapt easily to the system. If you yearn for all the string-handling actions of a good BASIC, then Chapter 8 contains a set of functions and procedures which will make string-handling considerably easier for you.

ARRAYS OF STRINGS

In BASIC, you are accustomed to being able to use string arrays, with assignments such as A$(5)="FRIDAY". In Oxford Pascal, a string array is an array of an array of CHAR. This sounds complicated until you realise that it's no more than a two-dimensional array. The sort of thing which in BASIC is written as A(3,4) is treated very similarly in Oxford Pascal, with the minor difference that one of the dimensions is a set of ASCII codes. Once you have defined your 'string array' name correctly, you can use a string array rather as you do in BASIC. You must remember, however, that the rules are rather more strict. Each name in the array, for example, will consist of the same number of characters, with spaces used for padding. There's no point, for example, in having a function like the BASIC LEN action, because the length of each string will be whatever you defined it to be! This can have advantages, however. If you want to swop the position of two strings in the array, there's no problem because the two are of identical types. You can also completely re-assign an entire array.

Look for example at Figure 4.5. This consists of a program which will fill an array with names (of up to 20 characters), reassign the array to a different variable name, and then print the lot out. We start by defining NAMES as a packed array of characters, 20 characters long. Two variables are then defined. Both of

```
 10  PROGRAM P4N5;
 20  TYPE NAMES=PACKED ARRAY[1..20] OF C
HAR;
 30  VAR NAME,NAMLST:ARRAY[1..10]OF NAME
S;
 40  N:INTEGER;
 50  BEGIN;
 60  FOR N:=1 TO 10 DO
 70    BEGIN;
 80    WRITE('NAME, PLEASE ');
 90    READLN(NAME[N]);
100    END;
110  PAGE;
120  NAMLST:=NAME;
130  FOR N:=1 TO 10 DO
140    BEGIN;
150    WRITELN(NAMLST[N]);
160    END;
180  END.
```

Figure 4.5 One way of defining a string array. Line 20 defines the string as an array of **CHAR**, line 30 defines **NAME** and **NAMLST** as an array of strings.

them are arrays of NAMES, in other words, string arrays in which each string consists of twenty characters. The definition in line 30 allows for up to 10 of these names in the arrays NAME or NAMLST. In the first loop, lines 60 to 100, the array NAME is filled with names that you type from the keyboard. The screen is then cleared, and the new array NAMLST is then assigned to NAME. This copies all of the names in the NAMES array into the array NAMLST. In BASIC, this could be done only by setting up a loop such as:

```
100 FOR N=1 TO 10
110 NAMLST(N)=NAME(N)
120 NEXT
```

The names in this array are then printed on the screen by using the loop in lines 130 to 160. Each name in the array is obtained by using its position number within square brackets; NAMLST[7] in Oxford Pascal is equivalent to NAME$(7) in BASIC.

A lot of BASIC programs depend on filling an array with values which are taken from a list. You might, for example, want to fill an array WEEK$ with names taken from a DATA list of weekdays, such as

```
20 FOR N=1 TO 5
30 READ WEEK$(N):NEXT
```

so that any day can be found by use of the array number. This is not quite so easy in Oxford Pascal, because in Pascal there is no direct equivalent of the BASIC READ and DATA instructions. These instructions are very wasteful of memory, because everything that you have in a DATA line in BASIC is stored in two places while the program is running. As usual in Pascal, when there is no action which is provided by the built-in functions, you have to do it for yourself. The action of READ...DATA in BASIC can be simulated by reading from one array into another. The program of Figure 4.6 illustrates this action in the form of a procedure which you can use for your own programs. This is the longest sample that we have looked at so far, and when your programs get to this length, a printer becomes a more pressing necessity. The problem is that by the nature of Pascal, you tend to have a lot of nested BEGIN and END; statements. If you can see these only on the screen, it's very difficult to be sure that each END corresponds to the correct BEGIN. Of course, if you planned the program correctly in the first place, you will have checked the nesting on paper. The problem arises, however, when you have been doing some editing, renumbering the lines, correcting mistakes and so on. At that stage, checking for an incorrectly placed END; on the screen alone can be rather a frustrating task. One thing that can make a Pascal program easier to read is indenting each new BEGIN or loop. In this way, sections which are 'compound statements', running as if they consisted of one single instruction, are set away from the left-hand side in a block. This makes it easier to see where the BEGIN and END of each block is located.

Returning to the program of Figure 4.6, the main program is in lines 260 to 340. Line 270 defines a string which is constructed like a DATA line in BASIC,

```
 10  PROGRAM P4N6;
 20  TYPE DAY=PACKED ARRAY[1..9]OF CHAR;
 30  WEEKLIST=PACKED ARRAY[1..41]OF CHAR
;
 40  VAR WEEK:WEEKLIST;
 50  NEWDAY:DAY;
 60  WEEKDAY:ARRAY[1..5] OF DAY;
 70  N:INTEGER;
 80  PROCEDURE FILLARRAY;
 90  VAR X,K,J,N:INTEGER;
 100   BEGIN;
 110   J:=0;
 120   FOR N:=1 TO 5 DO BEGIN
 130    K:=1;
 140    J:=J+1;
 150    WHILE WEEK[J]<>',' DO BEGIN
 160      NEWDAY[K]:=WEEK[J];
 170      K:=K+1;
 180      J:=J+1;
 190      END;
 200        FOR X:=K TO 9 DO BEGIN
 210        NEWDAY[X]:=' ';
 220        END;
 230       WEEKDAY[N]:=NEWDAY;
 240     END;
 250   END;
 260  BEGIN;
 270   WEEK:='MONDAY,TUESDAY,WEDNESDAY,T
HURSDAY,FRIDAY,';
 280   FILLARRAY;
 290   FOR N:=1 TO 5 DO
 300    BEGIN;
 310    WRITELN('DAY ',N:1,' IS ',WEEKDA
Y[N]);
 320     END;
 340  END.
```

Figure 4.6 Simulating the action of **READ...DATA** in BASIC. You really should not need this action in Pascal, but until you get used to working with Pascal, it's comforting to be able to use it!

except that *there must be a comma following the last item.* This string has been precisely dimensioned in line 30. The procedure FILLARRAY will then read the items that are separated by commas from the string WEEK, and put each item into another array, padding out with spaces as necessary. Lines 290 to 320

simply prove that this has been correctly done by printing out the (string) array items.

The important part, then, is the procedure FILLARRAY, which is in lines 80 to 250. Line 90 declares a set of integer variables which are local to this procedure. Integer J is used to count characters from the long string WEEK, and integer K is used to count the characters into each string of the new string array, WEEKDAY. Integer N is used to count the string array items, five in this case, and integer X is used to count the spaces which will be used to pad each string out to the correct length. Before the action starts, J is set to zero, and the main loop starts in line 120. This consists of a FOR N:=1 TO 5 loop, because we want to create five items of a string array. For each one, we need to start with K=1, counting the characters into the array, and incrementing J. On the first pass through this loop, J has to be incremented to avoid having a zero count; on later passes, incrementing J on each pass has the effect of skipping over the comma. Great care needs to be taken when you are designing programs which shift characters between arrays, because it is easy to over-fill an array. This will not show up when you compile, but will cause a run-time error. A run-time error is more awkward to sort out, because the cause is often difficult to see, and you may have to edit in a line (such as a WRITELN(J,K); instruction) for diagnosis of the problem. This means re-compiling, running, altering and so on until the fault is corrected.

The transfer of characters from the 'string variable' WEEK to a temporary variable, NEWDAY, is done by another loop. This loop is one that we have not come across before, and if you have programmed only the C-64, will be new to you. It is a WHILE type of loop, which is used in the BASIC of some machines, notably the Amstrad CPC464. In a loop of this type, a test is made at the start of the loop, and the loop is carried out only if the test succeeds. Unlike the FOR N:= 1 TO 5 type of loop, a WHILE loop (and its companion, the REPEAT loop) does not depend on a count-up or countdown, only on a test. In this case, the test is that the character which has been sampled in array WEEK is not a comma. While the character is not a comma, the loop continues. What is done in this loop is placed between the BEGIN at the end of line 150 and the END; in line 190. The character is copied from one string to the other, and the counter variables J and K are both incremented. When the value of J causes a comma to be read, however, the program jumps from line 150 to line 200, following the loop. At this point, another FOR loop is used, counting from K, the (incremented) value of place number in NEWDAY to 9, the limit of size. This loop places in as many spaces as are needed to make the string up to its correct length. This may not be necessary, since the length of the string is fixed by its definition, but it may be needed to clear the memory that is used for the string. The assignment in line 230 can then be carried out. This assigns the NEWDAY that has been read from string WEEK into the array WEEKDAY.

From then on, the pattern repeats. With N=2, K is reset to 1, and J is incremented so as to skip over the comma in the string WEEK. Another set of char-

acters up to a comma is copied, padded out and assigned. Note that the string WEEK must end with a comma. If this is not done, there is no end marker, and the program will try to continue reading characters. This will over-fill the array, and you will get an error message when you run. All in all, you might prefer, if you have only a few items to use, simply to assign the items directly, using lines like

```
90 WEEKDAY[1]:='MONDAY '
```

and remembering the spaces. For larger sets of items, however, this procedure can be useful—after all, you don't have to think it out afresh each time. The best answer of all, however, when you have to assign an array with a set of ready-made items like this is to read the items from a disk file. We'll look at disk files later in this chapter.

SETS

A *set* is yet another of the 'structured' data types of Pascal which has no separate life of its own in BASIC. In Pascal, a set is a collection of items, which may be integers or names, but *not* real numbers. You define a set, and assign it to a variable name, by using the statement SET OF, and following it with the items enclosed by square brackets. For example, if you want your set to consist of the numbers 1 to 8 inclusive, you can use

```
50 VAR OCTET:SET OF [1..8];
```

to define the set. You then have to assign the set, using something like

```
100 OCTET:=[1..8];
```

—you can use a smaller range of numbers if you like, but not anything that falls outside the set.

The advantage in using a set like this is that it allows you to test if quantities belong to the set. In the set OCTET, for example, any integer from 1 to 8 belongs, and anything less than 1 or more than 8 does not. This avoids the complicated tests that we so often need in BASIC, because all that we need to check if something is in a set is to use the word IN for the test. We can use a line such as

```
IF X IN OCTET THEN
```

to find if the value of X is something that lies in the range that we have specified for OCTET. We can then use the IF test to decide what to do. This looks like a convenient place for dealing with a set example, and to take in IF tests and the REPEAT...UNTIL loop as part of the bargain. Later, we'll look at how to phrase the test for something not being in the set.

Figure 4.7 shows a simple number-guessing program. The idea is to guess a number which must lie between 1 and 8. The program has to include rejection of any number outside the correct range, which will be done by using a set. It must also check whether your guess is correct or not, and print suitable messages. The program loops round and terminates when the number 999 is entered as a guess. As usual, the program has been constructed round a simple

```
10  PROGRAM P4N7;
20  VAR OCTET:SET OF 1..8;
30  X,Y:INTEGER;
40  PROCEDURE COMPAREGUESS;
50    BEGIN;
60    IF (X=Y) THEN
70    WRITELN('WELL DONE')
80    ELSE WRITELN('YOU MISSED');
90    END;
100  PROCEDURE NOTGOOD;
110   BEGIN;
120   WRITELN('RANGE IS 1 TO 8 ONLY');
130   WRITELN('PLEASE TRY AGAIN');
140   END;
150  BEGIN;
160  PAGE;
170  OCTET:=[1..8];
180   REPEAT
190    WRITELN('YOUR GUESS?');
200    READLN(X);
210    Y:=RANDOM MOD(8)+1;
220      IF X IN OCTET THEN
230      COMPAREGUESS
240      ELSE NOTGOOD;
250   UNTIL (X=999);
270  END.
```

Figure 4.7 Using a **SET**, in this case a set of integer numbers. This allows you to check if a reply is in the correct range without stopping the program if it is not.

core in lines 150 to 260. The screen is cleared, and OCTET is assigned as meaning the set 1..8. Lines 180 to 250 then form a loop of the REPEAT...UNTIL type. As the name suggests, this loop will keep repeating until a test at the end of the loop succeeds. The fact that the test is at the end of the loop means that the loop actions will always be carried out at least once, because no test is made until the end of the loop. Compare this with the WHILE loop, in which the test was made at the start of the loop.

If you are accustomed to the BASIC of the Commodore 64 and have used no other machine, the extension of the IF test with ELSE will be new. The idea is that you can program an alternative without having to use a new IF line. For example, you can have something like

```
100 IF X=Y THEN
110 (whatever it is)
120 ELSE
130 (something else);
```

so that if X=Y then the action in line 110 is carried out, if X is not identical to Y, then the action in line 130 is carried out. The test must be one which has a TRUE or FALSE outcome, the type of test that is called Boolean. One thing that you need to be careful of here is that you don't have so many ELSE lines that you lose track of what you are trying to do! The other is a matter of syntax. You must not have any semicolon on the line that comes just before the ELSE part of a test, even though you may use several lines between IF and ELSE. This is one of the few places where semicolons are forbidden, so it should be reasonably easy to remember.

Returning to the program, then, you are asked for a guessed number which you enter in line 200. Line 210 then demonstrates one of the extensions to standard Pascal, especially for the C-64. This allows a number to be generated at random (or almost at random). MOD is used to ensure that the result is in the correct range. The action of MOD is to divide by the number that follows, in brackets, and give the remainder. For example, 20 MOD 8 is 4, because 20/8 is 2 with a remainder of 4. RANDOM MOD(8) means that a random number is divided by 8 and the remainder used. This remainder can be any number between 0 and 7. Adding 1 to this gives a range of 1 to 8, which is what we need for our random number. Now we have to check the number that we have entered. First of all, we check to find if it is in the permitted range of 1 to 8. This is done by line 220. If the entered number X is in the range OCTET, then procedure COMPAREGUESS is used to check the guess with the random number. If the value that you entered was outside the OCTET range, then procedure NOTGOOD is run. All of these decisions are made by lines 220 to 240. Try this action in BASIC to see how it looks! The action then repeats until a value of 999 is entered. I have placed brackets around the test quantities such as (X=999) to emphasise that each of these must be TRUE or FALSE. The brackets are not absolutely necessary in this example, but they need to be used if you are carrying out more than one test in a line.

The actions that are carried out by the procedures must now be looked at. The simplest one is NOTGOOD, which is the one that is selected if X is not in OCTET. This procedure simply writes the message about selecting from the correct range. The other procedure, COMPAREGUESS, is the one that is used when your number is within the correct range. If your guess X equals the random number Y, then it's congratulations. If the two are not equal, the ELSE part swings into action, with the other message. Obviously, if you wanted to keep a score of points, you could include the score increment action in a line 65.

OTHER SET ACTIONS

The IN test for an item being a member of a set is very useful, but it is just one of the tests that can be made on a set. The other tests are used to compare sets to find out about items that two sets may have in common. The tests are intersec-

tion, union, difference, equality and inequality. Each of these uses a mathematical sign, and you have to be careful to distinguish between the use of these signs for arithmetic and their use for set comparison. The * sign means intersection, and SET1*SET2 means *all* of the items that appear in both sets. For example, if SET1 consists of letters A,B,C,D and SET2 of letters B,D,E,F then SET1*SET2 consists of B,D; all of the letters that appear in both sets. A comparison like this can be made only if the items in the sets are compatible, meaning types that can be compared in this way. This is one good reason for not allowing REAL numbers to be used, because a REAL is always stored as an approximation, and tests like this could not always be valid. A test such as SET1*SET2 should be read (and thought of) as SET1 AND SET2, meaning the items which appear in SET1 *and* SET2.

The + sign is used to mean 'union', conveniently read as OR. SET1+SET2 means the items which appear in one set *or* the other *or* both. In the example of letters, SET1+SET2 would consist of A, B, C, D, E and F. These are all of the items in the two sets. The − sign is more tricky. SET1−SET2 means all of the items that appear in SET1 which are not in SET2. This one can be turned the other way round, into SET2−SET1, to mean all of the items in SET2 which are not in SET1.

All of these comparisons will give Boolean results, that is either TRUE or FALSE. If you have not worked with sets before—and you certainly will not have done with Commodore 64 BASIC—you need to be careful how you go about programming set actions like these. Take a look at the example in Figure 4.8. This defines two sets, FRIENDS and NEIGHBOURS, which consist of a set of names, six in all. The names FRIENDS and NEIGHBOURS are then assigned in lines 40 and 50, so that FRIENDS means SMITH, BLACK and NORTH and NEIGHBOURS means JONES, BLACK, NORTH and SOUTH. The rest of the program then shows what happens when various set actions are carried out. The test in line 60 gives the result TRUE. This is because the names BLACK and NORTH are the two names which are common to both groups—they are friends *and* neighbours. Line 70, however, gives FALSE. This is because BLACK is not the only friend and neighbour; there are two people of this description. This is where you have to be careful because it is very tempting to assume that this statement should be true. The correct way of getting a TRUE statement from the idea that BLACK is one of the people who are friends and neighbours is in line 80. By using BLACK IN FRIENDS*NEIGHBOURS, we are asking if the name BLACK is one of the names which are common to the two groups FRIENDS and NEIGHBOURS. Note the difference in the way that BLACK is enclosed. When you are testing by using = (or any other signs like <=,>=,<> and so on), the item name has to be enclosed in square brackets. When you use the IN type of test, the item does *not* need brackets. If you get this wrong, you will get the message to the effect that WRITELN can only write integer, real, char, Boolean or string. The name BLACK is none of these—it's part of a set. When BLACK is enclosed in square brackets, you are comparing a set item with

```
 10  PROGRAM P4N8;
 20  VAR FRIENDS,NEIGHBOURS:SET OF (SMIT
H,JONES,BLACK,WHITE,NORTH,SOUTH);
 30  BEGIN;
 40  FRIENDS:=[SMITH,BLACK,NORTH];
 50  NEIGHBOURS:=[JONES,BLACK,NORTH,SOUT
H];
 60  WRITELN([BLACK,NORTH]=FRIENDS*NEIGH
BOURS);
 70  WRITELN([BLACK]=FRIENDS*NEIGHBOURS)
;
 80  WRITELN(BLACK IN FRIENDS*NEIGHBOURS
);
 90  WRITELN(SOUTH IN NEIGHBOURS-FRIENDS
);
 100  WRITELN([SMITH]=FRIENDS-NEIGHBOURS
);
 120  END.
```

Figure 4.8 Actions on **SETS**. The results are not what you might expect unless you know a bit about sets already.

another of the same type, and the comparison will give TRUE or FALSE, which is type Boolean. The compiler's honour is then satisfied. The different syntax for IN arises because the IN test is for one single item being in a set; whereas you might use [BLACK,WHITE] as a test for AND, OR or whatever else.

This makes line 90 easier to explain. The test is for SOUTH in NEIGHBOURS-FRIENDS. Now the set of NEIGHBOURS-FRIENDS is JONES and SOUTH. If we tried [SOUTH]=NEIGHBOURS-FRIENDS we would get the result FALSE, because SOUTH is just one part of NEIGHBOURS-FRIENDS. By using SOUTH IN we get the TRUE result, because SOUTH is one of the two which are defined by NEIGHBOURS-FRIENDS. Note that SMITH is not considered in this comparison—there is no meaning attached to the idea of a negative name! Finally, in line 100, we try FRIENDS-NEIGHBOURS. This gives SMITH only—remember that SOUTH can't be counted in this, because we are looking at the names which are in FRIENDS but not in NEIGHBOURS. SOUTH is not in FRIENDS, so the name can't appear in this set.

Normally, of course, we would not use WRITELN to show these comparisons. The more usual thing would be to include these in IF tests, so that the program could then go on to do something else. You might, for example, like to send one Christmas greeting to neighbouring friends, and another to neighbours who were not particularly friendly. Taking another example, you might want to send one list of items for sale to Commodore 64 owners who do not have a printer or a disk system, another list to those who had a disk system but

no printer, another to owners of a printer but no disk system and so on. Actions like these have to be programmed very tediously in BASIC, and you don't quite realise how much freedom Pascal gives you until you try it.

5

MENUS, CHOICES AND FILES

The BASIC of the Commodore 64 does allow the simplest method of programming menus, using the ON K GOSUB type of command. Since a menu is a very common feature of a lot of programs, it's time that we took a look at how such a system can be programmed in Oxford Pascal. The key to simple menu programming is the CASE statement. Suppose that you have a list of items on your menu, with each item numbered in the usual way. You then use READLN with a variable name to input a number. Suppose, for example, that your variable name is REPLY. You can then program

```
CASE REPLY OF
1:(first action);
2:(second action);
```

and so on. The variable REPLY will have values which are numbers such as 1, 2, 3 and so on. This allows CASE to select the command which appears after the same number in the list that follows CASE OF. In a real program, each of these options would consist of a procedure name, or a set of procedures. For an illustration, we can substitute simple WRITELN statements, as in Figure 5.1.

In this example, the word REPLY has been defined as an integer, and it will be used for the CASE statement. The screen is cleared in line 40, and a title is printed centred by using a suitable field number in line 50. The menu is then printed, with a number allocated to each item. You are asked to choose by number, and then your number choice is assigned to the variable REPLY in line 140. The CASE part runs from line 150 to 210. The word CASE must be followed by the variable name REPLY, and then OF, with no semicolon. What follows lists the numbers and actions. Each number is followed by a colon, then the action or actions that must be carried out. The set of CASE actions must end with an END;, and the whole program ends as usual with the END that is followed by a full stop. Each choice has simply caused a phrase to be printed in this

```
10  PROGRAM P5N1;
20  VAR REPLY:INTEGER;
30  BEGIN;
40  PAGE;
50  WRITELN('THE MENU':24);
60  WRITELN;
70  WRITELN('1. START FILE.');
80  WRITELN('2. ADD TO FILE.');
90  WRITELN('3. DELETE ITEM.');
100  WRITELN('4. AMEND ITEM.');
110  WRITELN('5. END PROGRAM.');
120  WRITELN;
130  WRITELN('PLEASE CHOOSE BY NUMBER')
;
140  READLN(REPLY);
150  CASE REPLY OF
160     1:WRITELN('START, THEN');
170     2:WRITELN('ADD, THEN');
180     3:WRITELN('DELETE THEN');
190     4:WRITELN('AMEND THEN');
200     5:WRITELN('END THEN');
210     END;
230  END.
```

Figure 5.1 Using **CASE OF** to carry out a menu choice action. Normally, each action would consist of a procedure, but **WRITELN** has been used here to illustrate the use of **CASE**.

example, because the aim is just to show what the CASE statement does and how it is programmed.

Now a menu of this type is certainly simple, but what happens if the user types a number which is not in the list? The answer is that you get a run-time error report, CASE ERROR, LINE 150, appearing on the screen, and the program stops. This is undesirable, so the obvious answer is to include a mugtrap, otherwise known as an error validation step. This means simply that you check to make sure that the range of REPLY is correct before you use it. We have already seen how we can make use of a SET for data checking, so we can modify this menu program to suit. This time (Figure 5.2), the name RANGE is used for the set of integers 1 to 5 that will be used for the menu selection. The variable is defined in line 30, and assigned in line 90. A procedure has been added which will print an error message if an incorrect selection is made; this is in lines 40 to 70. The test for range is now made in line 210, and at first sight it looks rather odd. The reason is that whatever follows IF must be Boolean, meaning that it must be TRUE or FALSE. Now if you write

```
IF REPLY NOT IN RANGE
```

```
10  PROGRAM P5N2;
20  VAR REPLY:INTEGER;
30  RANGE:SET OF 1..5;
40  PROCEDURE MESSAGE;
50  BEGIN;
60  WRITELN('INCORRECT CHOICE')
70  END;
80  BEGIN;
90  RANGE:=[1..5];
100  PAGE;
110  WRITELN('THE MENU':24);
120  WRITELN;
130  WRITELN('1. START FILE.');
140  WRITELN('2. ADD TO FILE.');
150  WRITELN('3. DELETE ITEM.');
160  WRITELN('4. AMEND ITEM.');
170  WRITELN('5. END PROGRAM.');
180  WRITELN;
190  WRITELN('PLEASE CHOOSE BY NUMBER')
;
200  READLN(REPLY);
210  IF NOT(REPLY IN RANGE)THEN MESSAGE
220   ELSE
230    CASE REPLY OF
240      1:WRITELN('START, THEN');
250      2:WRITELN('ADD, THEN');
260      3:WRITELN('DELETE THEN');
270      4:WRITELN('AMEND THEN');
280      5:WRITELN('END THEN');
290    END;
310  END.
```

Figure 5.2 How to prevent a **CASE** statement run-error if an incorrect number is selected.

then what follows IF is a variable name, and then you ask if NOT is in RANGE. The compiler gives up at this example of human awkwardness. What you have to do is to rearrange your question. Normally, you would want REPLY IN RANGE. To get the opposite of this, you enclose REPLY IN RANGE with brackets. This will ensure that it is treated first, giving TRUE or FALSE. Putting NOT in front of it will then turn any TRUE into FALSE and any FALSE into TRUE. The IF statement can then cope with this, with the effect that if the number is not in the correct range, then procedure MESSAGE will be carried out. By following this with ELSE, we ensure that if procedure MESSAGE is not carried out (the number was acceptable), then the CASE statement will oper-

ate. The net result is that if the number is acceptable, the program works just as it did. If the number is unacceptable, then the message is printed and the program ends.

In many cases, though, that's still not what we want. What we would like is to have the menu repeated until we enter a number that is suitable. In BASIC you have to use a GOTO to achieve this, but in Pascal, the obvious way is to use the REPEAT instruction. This is illustrated in Figure 5.3, in which the selection is

```
10  PROGRAM P5N3;
20  VAR REPLY:INTEGER;
30  RANGE:SET OF 1..5;
40  PROCEDURE MESSAGE;
50  BEGIN;
60  WRITELN('INCORRECT CHOICE');
70  WRITELN('PLEASE TRY AGAIN 1-5 ');
80  END;
90  BEGIN;
100  RANGE:=[1..5];
110  PAGE;
120  WRITELN('THE MENU':24);
130  WRITELN;
140  WRITELN('1. START FILE.');
150  WRITELN('2. ADD TO FILE.');
160  WRITELN('3. DELETE ITEM.');
170  WRITELN('4. AMEND ITEM.');
180  WRITELN('5. END PROGRAM.');
190  WRITELN;
200  WRITELN('PLEASE CHOOSE BY NUMBER')
;
210  REPEAT
220  READLN(REPLY);
230  IF NOT(REPLY IN RANGE)THEN MESSAGE
240  UNTIL REPLY IN RANGE;
250    CASE REPLY OF
260      1:WRITELN('START, THEN');
270      2:WRITELN('ADD, THEN');
280      3:WRITELN('DELETE THEN');
290      4:WRITELN('AMEND THEN');
300      5:WRITELN('END THEN');
310    END;
330  END.
```

Figure 5.3 Using a **CASE** menu within a **REPEAT** loop so that an incorrect entry is notified and entry repeated until a correct entry is received. The **REPEAT** loop consists of all the lines between **REPEAT** and **UNTIL**.

repeated until a choice in the correct range is made. This time, the program does not end when an incorrect choice is made. The message is printed by the procedure, and the REPEAT loop ensures that the choice can be made again until the number REPLY lies in the correct range.

OTHER CASES

The control for CASE does not have to be an integer, but Oxford Pascal does not allow some of the features of other varieties of Pascal, such as having a range like 0..31 as a choice. You can, however, use a character to control a CASE, as is illustrated in Figure 5.4. In this example, the variable REPLY is of

```
10  PROGRAM P5N4;
20  VAR REPLY:CHAR;
30  BEGIN;
40  WRITELN('TYPE A LETTER');
50  READLN(REPLY);
60  CASE REPLY OF
70     'A','E','I','O','U':WRITELN(REPLY
,' IS A VOWEL');
80    END;
100   END.
```

Figure 5.4 Using **CASE** with a letter entry. Any data type except a real number can be used in a **CASE** statement.

type CHAR, meaning a letter, and the one and only CASE statement tests more than one possibility in a single line. The letters that are tested for are the vowels, and each is typed surrounded by apostrophes, and separated by commas. If you type a vowel in response to the prompt in line 40, then, this will be identified as one of the items in the CASE line, and the letter will be printed, with the message IS A VOWEL following it. Because this is a simple example, it will stop with a CASE error message if you enter a consonant. You can, however, use CASE with any data type you like, except real numbers as usual. Most versions of Pascal allow you to end a CASE list with OTHERWISE, followed by some action statement, so that you can cater for items which are not in the CASE list. Oxford Pascal does not possess this useful let-out, and you must therefore be careful to see that something sensible happens no matter what is entered as a reply.

The identifying parts of a CASE do not have to be integers or single characters, they can be any type except real. They can, for example, be a type that

you have defined for yourself. This can look very promising when you see an example like Figure 5.5. The type SEASONS is declared as SPRING, SUMMER, AUTUMN, WINTER, and the variable SEASON is declared as one of the type SEASONS. This means that we can use a CASE statement with CASE SEASON OF, and follow this by lines that use the season names that

```
 10  PROGRAM P5N5;
 20  TYPE SEASONS=(SPRING,SUMMER,AUTUMN,
WINTER);
 30  VAR SEASON:SEASONS;
 40  BEGIN;
 50  SEASON:=SUMMER;
 60  CASE SEASON OF
 70    SPRING:WRITELN('MARCH, APRIL,MAY.
');
 80    SUMMER:WRITELN('JUNE,JULY,AUGUST.
');
 90    AUTUMN:WRITELN('SEPTEMBER, OCTOBE
R, NOVEMBER.');
 100    WINTER:WRITELN('DECEMBER, JANUAR
Y, FEBRUARY.');
 110    END;
 130  END.
```

Figure 5.5 Using **CASE** with a defined type. The variable name **SEASON** must be of type **SEASONS**, however; it can't be a string variable entered from the keyboard.

have been defined in the type declaration. The assignment in line 50 of Figure 5.5 will cause the selection of SUMMER, and so the words JUNE, JULY, AUGUST will be printed. The snag is that the assignment is fixed, there's no way of making the program print out any other months except by altering line 50. If you try to put a READLN in this position, you come up against the persistent problem of incompatible types. A READ or READLN can be used only for the standard data types. Even if you define a string variable (as a packed array of CHAR, remember) and read this, you cannot assign it to a SEASON name because, once again, these are incompatible types. This problem is skipped over very lightly, and sometimes not even mentioned, in most books on Pascal. If you want to enter strings and have them assigned in the program, you need to avoid sets and CASE statements. Both sets and CASE statements work only with what the compiler classes as 'scalar' types, meaning the built-in types of integer, Boolean and char, never real. The *structured* data types such as arrays (including strings) are excluded from this. You need to be clear about this point, because you can waste a lot of time trying to enter string values and test

them with IN, or use them in CASE statements. The compiler will refuse to accept such programming, and will deliver various error messages, depending on what syntax you are using. If you must assign a defined data type, then you have to do so indirectly, as the program in Figure 5.6 shows.

```
 10  PROGRAM P5N6;
 20  TYPE SEASONS=(SPRING,SUMMER,AUTUMN,
WINTER);
 30  VAR SEASON:PACKED ARRAY[1..6]OF CHA
R;
 40  MONTHS:SEASONS;
 50  TEST:BOOLEAN;
 60  BEGIN;
 70  REPEAT;
 80    TEST:=TRUE;
 90    WRITELN('WHICH SEASON?');
 100   READLN(SEASON);
 110   IF SEASON='SPRING' THEN MONTHS:=S
PRING
 120   ELSE IF SEASON='SUMMER' THEN MONT
HS:=SUMMER
 130   ELSE IF SEASON='AUTUMN' THEN MONT
HS:=AUTUMN
 140   ELSE IF SEASON='WINTER' THEN MONT
HS:=WINTER
 150   ELSE BEGIN;
 160     WRITELN('ILLEGAL ENTRY');
 170     TEST:=FALSE;
 180     END;
 190  UNTIL TEST=TRUE;
 200    CASE MONTHS OF
 210      SPRING:WRITELN('MARCH, APRIL,
MAY.');
 220      SUMMER:WRITELN('JUNE, JULY, AU
GUST.');
 230      AUTUMN:WRITELN('SEPTEMBER, OCT
OBER, NOVEMBER');
 240      WINTER:WRITELN('DECEMBER, JANU
ARY,FEBRUARY');
 250    END;
 270  END.
```

Figure 5.6 The rather clumsy method that is needed to identify a **CASE** type with an entered word. This is an action which is much simpler in BASIC.

In this example, you can enter a string, which is the name of a season, and get a list of the (supposed) months of that season. The variable SEASON is defined in the usual way as a packed array of CHAR, and the type SEASONS as the four names of the seasons. The variable MONTHS is of type SEASONS, and TEST is of type BOOLEAN. This means that MONTHS can take only the values of SPRING, SUMMER, AUTUMN or WINTER, and TEST can take only the values TRUE or FALSE. The main program then begins with a loop which will take a string input and test it. The input step is at line 100, within a REPEAT loop in which TEST is set to the value TRUE. Lines 110 to 140 test the input to see if it matches the name of a season. In each test, if the name matches, then MONTHS is assigned to a season name. This is possible because MONTHS is of type SEASONS, whereas SEASON is an array. The routine which starts in line 150 will act if none of the words that has been tested matches the word that has been input. In this loop, TEST is set to FALSE, so that the loop has to repeat. If a match was found earlier, this set of lines will not run, TEST is still TRUE, and there is no repeat. Once this validation has been carried out, the CASE statement can be used. This employs the season names directly, because, once again, the variable type which follows CASE matches the type that is used in the tests. Pascal, as I have said, is very fussy about data types, and will not allow you to mix them. If you encounter problems in using assignments or in CASE statements, you might want to look back at this example!

MORE STRUCTURED TYPES

The Oxford Pascal manual for the Commodore 64 refers in several places to 'textfiles', and it would be easy to jump to the conclusion that these could be used like string variables. In resident mode on the Commodore 64 version of Oxford Pascal, however, a textfile simply means a set of ASCII codes which can be entered from the keyboard, viewed on the screen as characters, or sent to a printer. The reason for using the name 'textfile' is that anything which is used in this way is treated more like a file than a variable. You don't, for example, need to declare that you will enter ten characters from the keyboard, or print six characters on the screen. Textfiles become more useful when we can save them on disk or load them from disk, and we'll look at that point in Chapter 9. We'll leave it until then, because using the disk compiler is rather a lengthy and tedious process and is not, of course, available for readers who have only the cassette system. For the moment, however, there are still plenty of topics to get to grips with. One of these is records, something that is not easy at the best of times, and more difficult if you have only ever programmed the Commodore 64 in BASIC. A record is a collection of items of data, which may all be of the same type or, more usually, of different types. What makes these items into parts of a record is that they are related. To take an example, suppose that you wanted to keep a record of membership of the local golf club. You would need

the name and the address for each member. These would be strings, packed arrays of CHAR. You might also want to keep year of birth (because juniors pay a reduced fee, and senior members pay only green fees), year of joining (members with ten or twenty years membership have special privilege years), and handicap. All of these last three items would be integers. There might also be an entry for fees due (a real number) and whether paid or not to date (a Boolean TRUE or FALSE).

Now all of this data constitutes a RECORD because for each person, the subject of the record, all the items belong together. It would not make much sense to keep a file of names, one of addresses, one of year of birth, and so on, and yet this is the way that we are often forced to keep such records in BASIC. The alternative in BASIC is often to pack all the data into one string of set length, and to make up a string array. Pascal allows you to define what will go into a record, and then to create an *array* of records. Obviously, the ultimate aim of such an array would be to record it on tape or disk, but that's something that we can leave until later. Another thing that we'll leave for later is the topic of using 'pointers' to locate records in memory.

We'll start by considering what we need to do to declare a record, using as an example the golf club illustration above.

The first action, as usual, is to declare any constants. In this illustration, we shall make the total membership of the club a constant, because this *is* just an example. By making this a small number, you can see how the program works without wearing out your typing finger(s). The important part, however, is what follows in the TYPE declaration. The name of the record is given as GOLF_CLUB. Notice that the _ character is the only one other than letters or digits that you can use in an identifier name. On the Commodore 64, you get this symbol by using the C= key along with the @ key. It's often very convenient for making names more readable, because you cannot have a space in a name. Unfortunately, because the Commodore 64 uses a different ASCII code for this symbol, it appears on the listing as the dollar sign, $. GOLF_CLUB is declared as a record, and what follows must be a list of the *fields* of the record, meaning the items that make up the record. I have typed these indented to make them more obvious. The name and address fields are both strings, but with different numbers of characters, so they have to be declared separately. The two years and the handicap are all integers, and can be declared on one line. The subscription amount is a real number—in a program which was seriously intended to keep records of this type, the amount of the subscription would be calculated from a formula, and printed when required, but in this example, I have made it an entered item. The letter R is of type CHAR, and will be used for a Y or N reply. Variable PAID is Boolean, because the subscription will either be paid or not—this club doesn't allow instalment payments! The end of the definition of the fields of this record is marked with the usual END; instruction.

Having declared what the RECORD will consist of, we now have to look at

```
 10  PROGRAM P5N7;
 20  CONST TOTAL=2;
 30  TYPE GOLF$CLUB=RECORD
 40    NAME:PACKED ARRAY[1..20]OF CHAR;
 50    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 60    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 70    SUBSCRIP:REAL;
 80    R:CHAR;
 90    PAID:BOOLEAN;
 100    END;
 110  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 120    J:INTEGER;
 130  BEGIN;
 140   FOR J:=1 TO TOTAL DO
 150    WITH MEMBER[J] DO
 160     BEGIN;
 170     PAGE;
 180     WRITE('NAME: ');
 190     READLN(NAME);
 200     WRITE('ADDRESS: ');
 210     READLN(ADDRESS);
 220     WRITE('YEAR OF BIRTH: ');
 230     READLN(BIRTH$YEAR);
 240     WRITE('YEAR OF JOINING: ');
 250     READLN(JOIN$YEAR);
 260     WRITE('HANDICAP: ');
 270     READLN(HCAP);
 280     WRITE('SUBSCRIPTION AMOUNT: ');
 290     READLN(SUBSCRIP);
 300     WRITE('PAID- Y OR N: ');
 310     READLN(R);
 320     IF R='Y' THEN PAID:=TRUE
 330     ELSE PAID:=FALSE;
 340     END;
 360  END.
```

Figure 5.7 Declaring a **RECORD** and entering data into it. Note that the **$** symbol in the listing should be the underscore, obtained by pressing the **C=** and @ keys together.

how we can put data into such records. The easiest method is to make an array of records, so that each 'item' of the array is a complete record. In line 110, then, we define variable MEMBER as being an array of 1..TOTAL of type GOLF_CLUB. This means that there will be two records in the array (because TOTAL was set to 2), and each record will consist of the items that have been declared in the RECORD TYPE definition. Letter J is declared as an integer to make use of a FOR loop, and the data entry part of the program starts in line 130. The first new item here is the syntax of dealing with a record, in line 150. The word WITH is a keyword, and it means that the list of actions which follows must be carried out for each record. The list of what has to be done with each record is marked out by the use, as usual, of BEGIN and END; words. In this example, the screen is cleared, using PAGE, for each record entry. You might want to elaborate this by printing a title, such as the record number. The items of the record are then prompted for, and entered. The entry is straightforward until we come to lines 300 to 330. You cannot enter a Boolean quantity with READLN, so the reply 'Y' or 'N' is entered, using character R and this reply is converted to Boolean TRUE or FALSE by the tests in lines 320, 330. Note that for convenience, any answer other than 'Y' will make the variable PAID equal to FALSE. The entry section which started with the BEGIN in line 160 ends in line 340, and then the complete main section of program ends in line 360.

Now when you compile this piece of program and run it, you can enter the items of the record, one by one. When you have pressed RETURN on the last item of a record, the screen clears, and the next record entry request comes up. Since we have specified only two records by our choice of TOTAL in line 20, you will see this happen only once, but it's enough to show what is happening. What we need to look at now is how we would display the records on the screen or print them out when we need a list of the club membership. This introduces quite a number of new ideas, because we need to be able to look at the records one by one, giving the club secretary time to check the details, and it's also time that we used instructions that affect the colour of text. Rather than showing only a new section of program, then, I have illustrated the whole program again in Figure 5.8. This retains the two records, but uses a string constant now to deliver a message. In addition, the record array is read and used to display details on the screen.

The string constant MESG is needed now because we have to stop the display after each record has been printed. If no 'press any key' step is put in, the records will flash by at a pace which is much too fast to read, and only the last record will be readable. The program follows the path of Figure 5.7 until line 360, which makes a space and then line 370 prints the PRESS ANY KEY message. Line 380 is not standard Pascal, but an extension for the Commodore 64 which is peculiar to Oxford Pascal. This allows the keyboard to be tested with GET and will loop round (because of the semicolon following DO) until a key is pressed. You need to be careful about this. If you do not terminate the WHILE loop with a semicolon following DO, then there is a danger of several instruc-

```
 10  PROGRAM P5N8;
 20  CONST TOTAL=2;
 30  MESG='PRESS ANY KEY TO PROCEED';
 40  TYPE GOLF$CLUB=RECORD
 50    NAME:PACKED ARRAY[1..20]OF CHAR;
 60    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 70    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 80    SUBSCRIP:REAL;
 90    R:CHAR;
 100    PAID:BOOLEAN;
 110    END;
 120  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 130    J:INTEGER;
 140  BEGIN;
 150   FOR J:=1 TO TOTAL DO
 160    WITH MEMBER[J] DO
 170     BEGIN;
 180     PAGE;
 190     WRITE('NAME: ');
 200     READLN(NAME);
 210     WRITE('ADDRESS: ');
 220     READLN(ADDRESS);
 230     WRITE('YEAR OF BIRTH: ');
 240     READLN(BIRTH$YEAR);
 250     WRITE('YEAR OF JOINING: ');
 260     READLN(JOIN$YEAR);
 270     WRITE('HANDICAP: ');
 280     READLN(HCAP);
 290     WRITE('SUBSCRIPTION AMOUNT: ');
 300     READLN(SUBSCRIP);
 310     WRITE('PAID- Y OR N: ');
 320     READLN(R);
 330     IF R='Y' THEN PAID:=TRUE
 340     ELSE PAID:=FALSE;
 350     END;
 360  WRITELN;
 370  WRITELN(MESG);
 380  WHILE GETKEY=CHR(0)DO;
 390  (*WAIT FOR ANY KEY*);
 400  SCREEN(6);(*BACKGROUND*)
 410  PEN(7);(*TEXT*)
```

```
 420  FOR J:=1 TO TOTAL DO
 430   WITH MEMBER[J] DO
 440    BEGIN;
 450    PAGE;
 460    WRITELN('MEMBER ':25,J);
 470    WRITELN;
 480    WRITE('NAME: ');
 490    WRITELN(MEMBER[J].NAME);
 500    WRITE('ADDRESS: ');
 510    WRITELN(MEMBER[J].ADDRESS);
 520    WRITE('YEAR OF BIRTH: ');
 530    WRITELN(MEMBER[J].BIRTH$YEAR);
 540    WRITE('JOINED IN: ');
 550    WRITELN(MEMBER[J].JOIN$YEAR);
 560    WRITE('HANDICAP: ');
 570    WRITELN(MEMBER[J].HCAP);
 580    WRITE('SUBSCRIPTION THIS YEAR: '
);
 590    WRITELN(MEMBER[J].SUBSCRIP:1:2);
 600    WRITE('SUBSCRIPTION ');
 610    IF MEMBER[J].PAID=TRUE THEN WRIT
ELN('PAID')
 620       ELSE WRITELN('NOT PAID');
 630   WRITELN;
 640  WRITELN(MESG);
 650  WHILE GETKEY=CHR(0)DO;
 660    END;
 680  END.
```

Figure 5.8 Illustrating how a set of records can be printed on the screen. The whole program has been printed so as to show how the sections fit together. You can load in the program of Figure 5.7 and edit it to obtain this version.

tions, up to the next semicolon, being repeated in the loop. CHR(0) is the character that is 'got' when no key is pressed. We then alter the conditions for display. Using SCREEN(6) sets the background colour to blue (the normal default colour), and PEN(7) sets the text colour to yellow. These colour numbers are the normal Commodore 64 colour numbers which are listed on page 62 of my copy of the Commodore 64 manual. There is also a BORDER instruction in Oxford Pascal which allows you to control this colour also. With the colours set, the familiar loop then starts. This time, the member number is put up on the top of the screen as a title, and the details of the member then follow. What is new here is how the field items are printed. You cannot use variable names such as NAME because there is no such variable declared. NAME is part of the record MEMBER, and for MEMBER[1], this has to be written as

MEMBER[1].NAME. In other words, you have to specify the record that you are using, then a dot, and then the name of the field item. Since an array is being used, we can use MEMBER[J] as the record specifier. If all this looks rather like too much typing for you, remember that you can type things like

```
WL.(MB.BIRTH_YEAR);
```

and then use the CHANGE facility of the editor to change this to

```
WRITELN(MEMBER[J].BIRTH_YEAR);
```

for each of the write lines. This can save a lot of time when you are typing a program with a lot of WRITE and WRITELN items. At the end of the printout, the Boolean variable PAID is read and used to decide whether to print PAID or NOT PAID following the word SUBSCRIPTION. Before the END; part of the routine, the 'PRESS ANY KEY' message is delivered, and the GETKEY routine is run to make the program hang up while you read the record.

You don't, of course, have to read the record in this way. You might simply want a list of members' names, or of names and addresses. You might want a list of names and handicaps, or a list of members whose subscriptions are not yet paid. This means that you will have to add a menu to the program, with all of the items like entering data and reading records made into procedures that can be called up by the menu. You might also want to have names sorted alphabetically in order of surname, or to have lists printed in order of ascending handicaps. We can't possibly, in a book of this length, go into every variation that can be done on this program outline, but having seen how one printout can be achieved, you will be able to arrange others for yourself. Routines such as sorting arrays into order are standard pieces of programming—one suitable routine is printed in an example in Chapter 6.

Another very useful feature is that you can call up any record, in any order, because the records are arranged in an array. Suppose, for example, that you use the section of program which is illustrated in line 430 onwards in Figure 5.9. This assumes that a variable K has been declared as being of type CHAR early in the program so that the GETKEY action can be used more extensively. This time, the readout section will be repeated until the spacebar is pressed—you might want to make use of another key for this, because it's easy to press the spacebar without thinking. All you need to do is to use a different ASCII code in the test, and alter the message line in line 670. The program gets a record number from you in line 480 and uses this to print a record as before. After the printout, the extended message is printed, and then the Pascal equivalent of the BASIC GET step is used. This assigns variable K (defined as CHAR) to GETKEY, and repeats the assignment until a key is pressed. This key is then tested in line 700. If it is the spacebar, then the program stops, but if it is any other key, the program loops round so that you can look at another record. This is a very useful feature to incorporate in a main menu, because when data has been loaded into the program, you don't want to let it go again until you have completely finished with it. In this example, because the records are held purely as arrays in memory, nothing can be recovered after the program has

ended, and starting the program again calls for more data to be entered. No validation has been carried out on the number that you can enter for a record number, either. If an out-of-range number is entered, the program will stop with an ARRAY INDEX ERROR message, and the data will, once again, be lost.

Suppose that you have a Commodore printer connected, and you want to write the file information to the printer? This is provided for, but the syntax is not completely straightforward until you have had some practice. First of all, a new variable has to be declared. The example in the Oxford Pascal manual uses the variable name of PRINTER, and this is ideal for the purpose. This variable name will be used to direct material to the printer, and the variable name is declared as type TEXT. If you have a printer, add the entry

```
PRINTER:TEXT;
```

to your variable list—you can use a line number 145 for this. You also have to add a new line

```
REWRITE(PRINTER,4,0);
```

just before the printout section, perhaps as line 485.

Now in the printout section of the program, where you have items such as WRITELN(MEMBER[J].NAME), you have to substitute WRITELN(PRINTER, MEMBER[J].NAME). This is easily done by using the CHANGE editing command in the form

```
CHANGE/WRITELN(/WRITELN(PRINTER,/,390-
```

which will make all of the lines from 390 onwards carry this change. When you use the program now with a printer connected, the entry of data is as it was before, but when you ask for a record and supply a record number, the result appears on the printer. You will find that the instructions about pressing a key to proceed will also appear unless you edit these lines back so that they appear on the screen only.

Now suppose that you want some records to appear on the screen only, but others on the printer only? You can deal with this by having a choice offered of printer or screen, and assigning the correct device numbers. If you have used the Commodore 64 BASIC commands for input and output, you will be familiar with the idea of device numbers and secondary address numbers. Oxford Pascal allows you to use these in much the same way. By using PRINTER,3,0 you are specifying screen output, whereas by specifying PRINTER,4,0 you are specifying the Commodore printer. If you are using a non-Commodore printer with a different device number, of course, you will have to substitute the correct device number and probably omit the secondary address number of 0. Figure 5.10 shows how these modifications would look when incorporated in the program, with the variable PRINTER:TEXT entered in the earlier part of the program. Line 500 asks PRINTER OR SCREEN? and then you are prompted for an entry of letters P or S. This entry uses the variable K which would be defined as CHAR in the early part of the main program. Lines 530 and 540 then assign the REWRITE instruction. This instruction is a file opening instruction, and PRINTER is the file name, with the numbers being used as

```
 10  PROGRAM P4N9;
 20  CONST TOTAL=2;
 30  MESG='PRESS ANY KEY TO PROCEED';
 40  TYPE GOLF$CLUB=RECORD
 50    NAME:PACKED ARRAY[1..20]OF CHAR;
 60    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 70    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 80    SUBSCRIP:REAL;
 90    R:CHAR;
 100    PAID:BOOLEAN;
 110    END;
 120  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 130    J:INTEGER;
 140    K:CHAR;
 150  BEGIN;
 160   FOR J:=1 TO TOTAL DO
 170    WITH MEMBER[J] DO
 180     BEGIN;
 190     PAGE;
 200     WRITE('NAME: ');
 210     READLN(NAME);
 220     WRITE('ADDRESS: ');
 230     READLN(ADDRESS);
 240     WRITE('YEAR OF BIRTH: ');
 250     READLN(BIRTH$YEAR);
 260     WRITE('YEAR OF JOINING: ');
 270     READLN(JOIN$YEAR);
 280     WRITE('HANDICAP: ');
 290     READLN(HCAP);
 300     WRITE('SUBSCRIPTION AMOUNT: ');
 310     READLN(SUBSCRIP);
 320     WRITE('PAID- Y OR N: ');
 330     READLN(R);
 340     IF R='Y' THEN PAID:=TRUE
 350     ELSE PAID:=FALSE;
 360     END;
 370  WRITELN;
 380  WRITELN(MESG);
 390  WHILE GETKEY=CHR(0)DO;
 400  (*WAIT FOR ANY KEY*);
 410  SCREEN(6);(*BACKGROUND*)
```

```
 420  PEN(7);(*TEXT*)
 430  REPEAT
 440  PAGE;
 450  WRITELN('LOOK AT RECORD':13);
 460  WRITELN;
 470  WRITELN('WHICH RECORD, PLEASE? ');
 480  READLN(J);
 490   WITH MEMBER[J] DO
 500    WRITE('NAME: ');
 510    WRITELN(MEMBER[J].NAME);
 520    WRITE('ADDRESS: ');
 530    WRITELN(MEMBER[J].ADDRESS);
 540    WRITE('YEAR OF BIRTH: ');
 550    WRITELN(MEMBER[J].BIRTH$YEAR);
 560    WRITE('JOINED IN: ');
 570    WRITELN(MEMBER[J].JOIN$YEAR);
 580    WRITE('HANDICAP: ');
 590    WRITELN(MEMBER[J].HCAP);
 600    WRITE('SUBSCRIPTION THIS YEAR: '
);
 610    WRITELN(MEMBER[J].SUBSCRIP:1:2);
 620    WRITE('SUBSCRIPTION ');
 630    IF MEMBER[J].PAID=TRUE THEN WRIT
ELN('PAID')
 640      ELSE WRITELN('NOT PAID');
 650    WRITELN;
 660    WRITELN(MESG);
 670    WRITELN('SPACEBAR STOPS DISPLAY'
);
 680     REPEAT K:=GETKEY
 690     UNTIL K<>CHR(0);
 700    UNTIL K=CHR(32);
 720  END.
```

Figure 5.9 How to select a record from a set of records—this program is an expanded version of Figure 5.8.

channel numbers for the path which the text is to take. This is, in fact, an example of a textfile in use. As usual, channel 3 means screen and channel 4 means hard copy. By defining PRINTER as being a text variable, we make it possible to transfer a file of text along the specified channel. The choice that is made here will therefore affect the record that is looked at, placing the details either on the screen or on the printer. Note that because only the READLN statements have been altered, the prompts will appear on the screen, but the

```
 10  PROGRAM P4N10;
 20  CONST TOTAL=2;
 30  MESG='PRESS ANY KEY TO PROCEED';
 40  TYPE GOLF$CLUB=RECORD
 50    NAME:PACKED ARRAY[1..20]OF CHAR;
 60    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 70    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 80    SUBSCRIP:REAL;
 90    R:CHAR;
 100    PAID:BOOLEAN;
 110    END;
 120  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 130    J:INTEGER;
 140    K:CHAR;
 150    PRINTER:TEXT;
 160  BEGIN;
 170   FOR J:=1 TO TOTAL DO
 180    WITH MEMBER[J] DO
 190     BEGIN;
 200     PAGE;
 210     WRITE('NAME: ');
 220     READLN(NAME);
 230     WRITE('ADDRESS: ');
 240     READLN(ADDRESS);
 250     WRITE('YEAR OF BIRTH: ');
 260     READLN(BIRTH$YEAR);
 270     WRITE('YEAR OF JOINING: ');
 280     READLN(JOIN$YEAR);
 290     WRITE('HANDICAP: ');
 300     READLN(HCAP);
 310     WRITE('SUBSCRIPTION AMOUNT: ');
 320     READLN(SUBSCRIP);
 330     WRITE('PAID- Y OR N: ');
 340     READLN(R);
 350     IF R='Y' THEN PAID:=TRUE
 360     ELSE PAID:=FALSE;
 370     END;
 380  WRITELN;
 390  WRITELN(MESG);
 400  WHILE GETKEY=CHR(0)DO;
 410  (*WAIT FOR ANY KEY*);
 420  SCREEN(6);(*BACKGROUND*)
 430  PEN(7);(*TEXT*)
```

```
 440  REPEAT
 450  PAGE;
 460  WRITELN('LOOK AT RECORD':27);
 470  WRITELN;
 480  WRITELN('WHICH RECORD, PLEASE? ');
 490  READLN(J);
 500  WRITE('PRINTER OR SCREEN?');
 510  WRITE('ENTER P OR S');
 520  READLN(K);
 530  IF K='P' THEN REWRITE(PRINTER,4,0)
 540     ELSE REWRITE(PRINTER,3,0);
 550   WITH MEMBER[J] DO
 560    WRITE('NAME: ');
 570    WRITELN(PRINTER,MEMBER[J].NAME);
 580    WRITE('ADDRESS: ');
 590    WRITELN(PRINTER,MEMBER[J].ADDRES
S);
 600    WRITE('YEAR OF BIRTH: ');
 610    WRITELN(PRINTER,MEMBER[J].BIRTH$
YEAR);
 620    WRITE('JOINED IN: ');
 630    WRITELN(PRINTER,MEMBER[J].JOIN$Y
EAR);
 640    WRITE('HANDICAP: ');
 650    WRITELN(PRINTER,MEMBER[J].HCAP);
 660    WRITE('SUBSCRIPTION THIS YEAR: '
);
 670    WRITELN(PRINTER,MEMBER[J].SUBSCR
IP:1:2);
 680    WRITE('SUBSCRIPTION ');
 690    IF MEMBER[J].PAID=TRUE THEN WRIT
ELN(PRINTER,'PAID')
 700      ELSE WRITELN(PRINTER,'NOT PAID
');
 710    WRITELN;
 720    WRITELN(MESG);
 730    WRITELN('SPACEBAR STOPS DISPLAY'
);
 740     REPEAT K:=GETKEY
 750     UNTIL K<>CHR(0);
 760    CLOSE(PRINTER);
 770    UNTIL K=CHR(32);
 790  END.
```

Figure 5.10 Selecting printer or screen for displaying records. Once again, this program consists of a reworking of previous examples. The **REWRITE** instruction is used in Pascal rather as **OPEN** is used in BASIC.

data will appear either on the screen or on the printer. The statement CLOSE(PRINTER) in line 760 closes the channel correctly, so that the next choice will work correctly. If you have a printer, this arrangement will be very useful to you.

SAVING THE FILE

When you use the first part of the golf club program (or your version of it) to create records, it's rather a shame if all the information goes to waste when the program ends. Now in the disk mode of Oxford Pascal, you can save complete records with a quite simple syntax, but this is not available in resident mode. You can, however, use the disk in rather the same way as you can use it with BASIC. If you use a cassette system, then you can also save and load sequential files with this. A file is saved by using the REWRITE instruction, followed by a set of quantities within brackets. The first quantity is a filename, which has to be a variable name that has been declared as being of type TEXT. The second quantity is a device number, using the Commodore standard numbers of 0 for screen, 1 for cassette, 4 for printer, and 8 for disk. The next item is another number, the secondary address number. For a disk unit this can be a channel number, and for a cassette unit it would be 0 for reading and 1 for writing. For a disk unit, a further item will be needed, such as a sequential file command, for example, '0:DATA,S,W' which would specify that this was a sequential file on drive 0, using the filename of DATA, and writing. If you use a disk unit and are not familiar with the syntax of files, then I respectfully suggest that you take a look at one of the many books available on the subject.

Figure 5.11 shows a program which will write the components of a record to a serial file. This is not so efficient as the correct file instructions of Pascal, but it is all that is available in resident mode, and it does at least allow you to check out a program before you go to all the hassle of using disk mode. Using resident mode is closer to using an interpreted language like BASIC in that you can run the program as soon as it has been compiled. In disk mode, mainly because of the slow action of the Commodore 64 disk system, you waste a lot of time if your program does not compile and run first time. It makes sense, then, to check out ideas in resident mode as far as possible, and then finally to change over to disk mode when you need to. This example and the one which follows show sequential disk filing because the very slow cassette recorder cannot seriously be considered for this type of use. In addition, substituting the cassette numbers for the disk numbers in the program always led to a machine hang-up when I tried it. Any attempt to list or break out of the hang-up caused a return to a clear screen, with READY showing, but not accepting commands in either Pascal editor or BASIC. The machine had to be switched off and on again to regain control.

In the disk writing program of Figure 5.11, then, the records are filled from

```
 10  PROGRAM P4N11;
 20  CONST TOTAL=2;
 30  MESG='PRESS ANY KEY TO PROCEED';
 40  TYPE GOLF$CLUB=RECORD
 50    NAME:PACKED ARRAY[1..20]OF CHAR;
 60    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 70    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 80    SUBSCRIP:REAL;
 90    R:CHAR;
 100    PAID:BOOLEAN;
 110    END;
 120  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 130    J:INTEGER;
 140    K:CHAR;
 150    DISK:TEXT;
 160  BEGIN;
 170   FOR J:=1 TO TOTAL DO
 180    WITH MEMBER[J] DO
 190     BEGIN;
 200     PAGE;
 210     WRITE('NAME: ');
 220     READLN(NAME);
 230     WRITE('ADDRESS: ');
 240     READLN(ADDRESS);
 250     WRITE('YEAR OF BIRTH: ');
 260     READLN(BIRTH$YEAR);
 270     WRITE('YEAR OF JOINING: ');
 280     READLN(JOIN$YEAR);
 290     WRITE('HANDICAP: ');
 300     READLN(HCAP);
 310     WRITE('SUBSCRIPTION AMOUNT: ');
 320     READLN(SUBSCRIP);
 330     WRITE('PAID- Y OR N: ');
 340     READLN(R);
 350     IF R='Y' THEN PAID:=TRUE
 360     ELSE PAID:=FALSE;
 370     END;
 380  WRITELN;
 390  WRITELN(MESG);
 400  WHILE GETKEY=CHR(0)DO;
 410  (*WAIT FOR ANY KEY*);
```

```
 420  SCREEN(6);(*BACKGROUND*)
 430  PEN(7);(*TEXT*)
 440  PAGE;
 450  WRITELN('SAVE RECORDS':26);
 460  WRITELN;
 470  REWRITE(DISK,8,2,'0:DATA,S,W');
 480  FOR J:=1 TO TOTAL DO
 490   WITH MEMBER[J] DO BEGIN
 500    WRITE('NAME: ');
 510    WRITELN(DISK,MEMBER[J].NAME);
 520    WRITE('ADDRESS: ');
 530    WRITELN(DISK,MEMBER[J].ADDRESS);
 540    WRITE('YEAR OF BIRTH: ');
 550    WRITELN(DISK,MEMBER[J].BIRTH$YEA
R);
 560    WRITE('JOINED IN: ');
 570    WRITELN(DISK,MEMBER[J].JOIN$YEAR
);
 580    WRITE('HANDICAP: ');
 590    WRITELN(DISK,MEMBER[J].HCAP);
 600    WRITE('SUBSCRIPTION THIS YEAR: '
);
 610    WRITELN(DISK,MEMBER[J].SUBSCRIP:
1:2);
 620    WRITE('SUBSCRIPTION ');
 630    IF MEMBER[J].PAID=TRUE THEN WRIT
ELN(DISK,'PAID')
 640       ELSE WRITELN(DISK,'NOT PAID');
 650  END;
 660  CLOSE(DISK);
 680  END.
```

Figure 5.11 Saving records on disk, using a 'textfile'. In resident mode, you have to use a **REWRITE** line which is very similar to the **OPEN** line for a serial file in BASIC.

the keyboard in the usual way. The routines are familiar until line 470, in which you find an extended REWRITE instruction. This uses the TEXT filename of DISK, and specifies the disk channel, number 8, the channel number of 2, and the usual sequential file data for writing. The items of the file are then written to the disk as shown. Note that the words PAID or NOT PAID are written, not the signal codes for TRUE or FALSE. The disk file is closed after all of the records have been written. If you have used disk files at all in BASIC, then this process should not feel too unfamiliar. The REWRITE instruction takes the place of the OPEN that is used in BASIC.

```
 10  PROGRAM P4N11;
 20  CONST TOTAL=2;
 30  MESG='PRESS ANY KEY TO PROCEED';
 40  TYPE GOLF$CLUB=RECORD
 50    NAME:PACKED ARRAY[1..20]OF CHAR;
 60    ADDRESS:PACKED ARRAY[1..40]OF CHA
R;
 70    BIRTH$YEAR,JOIN$YEAR,HCAP:INTEGER
;
 80    SUBSCRIP:REAL;
 90    R:CHAR;
 100    PAID:BOOLEAN;
 110    END;
 120  VAR MEMBER:ARRAY[1..TOTAL]OF GOLF$
CLUB;
 130    J:INTEGER;
 140    K:CHAR;
 150    DISK:TEXT;
 160  PAYMENT:PACKED ARRAY[1..8]OF CHAR;
 170  BEGIN;
 180  RESET(DISK,8,2,'0:DATA,S,R');
 190   FOR J:=1 TO TOTAL DO
 200    WITH MEMBER[J] DO
 210     BEGIN;
 220     PAGE;
 230     READLN(DISK,NAME);
 240     READLN(DISK,ADDRESS);
 250     READLN(DISK,BIRTH$YEAR);
 260     READLN(DISK,JOIN$YEAR);
 270     READLN(DISK,HCAP);
 280     READLN(DISK,SUBSCRIP);
 290     READLN(DISK,PAYMENT);
 300  IF PAYMENT='PAID    'THEN PAID:=TR
UE
 310  ELSE PAID:=FALSE;
 320     END;
 330  CLOSE(DISK);
 340  WRITELN;
 350  WRITELN(MESG);
 360  WHILE GETKEY=CHR(0)DO;
 370  (*WAIT FOR ANY KEY*);
 380  SCREEN(6);(*BACKGROUND*)
 390  PEN(7);(*TEXT*)
 400  PAGE;
```

```
 410  WRITELN('READ RECORDS':26);
 420  WRITELN;
 430  FOR J:=1 TO TOTAL DO
 440   WITH MEMBER[J] DO BEGIN
 450    WRITE('NAME: ');
 460    WRITELN(MEMBER[J].NAME);
 470    WRITE('ADDRESS: ');
 480    WRITELN(MEMBER[J].ADDRESS);
 490    WRITE('YEAR OF BIRTH: ');
 500    WRITELN(MEMBER[J].BIRTH$YEAR);
 510    WRITE('JOINED IN: ');
 520    WRITELN(MEMBER[J].JOIN$YEAR);
 530    WRITE('HANDICAP: ');
 540    WRITELN(MEMBER[J].HCAP);
 550    WRITE('SUBSCRIPTION THIS YEAR: '
);
 560    WRITELN(MEMBER[J].SUBSCRIP:1:2);
 570    WRITE('SUBSCRIPTION ');
 580    IF MEMBER[J].PAID=TRUE THEN WRIT
ELN('PAID')
 590       ELSE WRITELN('NOT PAID');
 600  WHILE GETKEY=CHR(0)DO;
 610  END;
 630  END.
```

Figure 5.12 A disk reading program which reads the records from the file that was created in Figure 5.11. The records are then printed.

You need to check now that you do actually have some files on the disk. To do so, modify the earlier version of the program so that it will read the disk files. The modified version is shown in Figure 5.12. This time, the read actions are from disk instead of from the keyboard, and the writing of records is to the screen. The disk system is opened for use with the RESET instruction in line 180, and the main difference this time is the use of R instead of W in the sequential file portion. The string variable PAYMENT has been defined as a character array to take the recorded message of PAID or NOT PAID from the disk. The screen reading action then follows along the paths that we have used before.

It is possible that the cassette version of Oxford Pascal would allow you to use cassette sequential files, in which case the REWRITE and RESET instructions would become

```
REWRITE(CAS,1,1); and
RESET(CAS,1,0);
```

with CAS declared as TEST in place of DISK. I cannot check this, and in any case, as I have said, if you seriously intend to use files you will also be using the disk system.

6

MORE ADVANCED RECORDS

The use of a simple record for keeping the individual fields of data together is useful, and particularly so when you can use the PUT instruction to save a complete record, rather than the individual fields of a record. This is possible only in disk mode, however, because in order to be able to use PUT (and GET) in this way, you need to declare a variable as a FILE of the record type. In resident mode, only a TEXT (short for textFILE) variable can be declared, and the REWRITE statement must use the standard disk filing commands, as we have illustrated. We shall take a brief look at disk mode in Chapter 9. Meanwhile, there are several points about the use of records that you need to investigate further before you start writing that long database program that you have always promised the pub darts league you would get down to.

Some of these points are illustrated in Figure 6.1. This is a database type of program which illustrates how complete records can be assigned. In BASIC, you cannot assign a group of variables to another group except by a set of individual assignments. This makes actions such as sorting very tedious, because each field of a record has to be represented by an array item or as part of a string. The illustration shows a Shell-Metzner type of sort routine used to sort a list of records in order of ascending handicap numbers. I have not used handicap numbers in the strict golfing sense here, so please don't write to say that the range should be from about +4 to −36! As always, we need to look at the main program first. This is in lines 690 to 750, and consists of the three procedures ENTER_NAME, SORT_NAME, and LIST_NAME, with the screen cleared before the listing. Incidentally, if you use command R on an uncompiled program, you will see that the Oxford Pascal compiler ignores characters following the eighth, so that these names become ENTER_NA, SORT_NAM and LIST_NAM respectively. You will have to be careful that none of your names, when cut down in this way, conflict with each other. Don't, for example, have names like NUMBER_OF_SPACES and NUMBER_OF_LINES, because these will not be taken as separate names.

```
 10  PROGRAM P6N1;
 20  CONST MAX=5;
 30  TYPE PERSON=RECORD
 40       SURNAME,FORENAME:PACKED ARRAY[
1..20]OF CHAR;
 50       AGE:INTEGER;
 60       HANDICAP:0..24;
 70  END;
 80  VAR MEMBER:ARRAY[1..MAX]OF PERSON;
 90      TEMP:PERSON;
 100     J,K:INTEGER;
 110     STRING:PACKED ARRAY[1..20]OF CH
AR;
 120  PROCEDURE ENTER$NAME;
 130  VAR J:INTEGER;
 140  BEGIN;
 150   FOR J:=1 TO MAX DO
 160    WITH MEMBER[J] DO BEGIN
 170     WRITE('SURNAME: ');
 180     READLN(SURNAME);
 190     WRITE('FORENAME: ');
 200     READLN(FORENAME);
 210     WRITE('AGE: ');
 220     READLN(AGE);
 230     WRITE('HANDICAP: ');
 240     READLN(HANDICAP);
 250    END;
 260  END;(*OF ENTER$NAME*)
 270  PROCEDURE SORT$NAME;
 280  VAR I,Y,J,X,Z,IT:INTEGER;
 290  BEGIN;
 300   Y:=1;
 310   WHILE Y<MAX DO Y:=2*Y;
 320  REPEAT
 330  Y:=TRUNC((Y-1)/2);
 340   IT:=MAX-Y;
 350   FOR I:=1 TO IT DO BEGIN;
 360    J:=I;
 370    REPEAT
 380     Z:=J+Y;
 390     IF MEMBER[Z].HANDICAP<=MEMBER[J
].HANDICAP THEN BEGIN
 400       TEMP:=MEMBER[Z];
 410       MEMBER[Z]:=MEMBER[J];
```

```
 420          MEMBER[J]:=TEMP;
 430          J:=J-Y;
 440          END
 450        ELSE J:=0;
 460         UNTIL J<=0;
 470      END;
 480    UNTIL Y=1;
 490    END;(*OF SORT*)
 500    PROCEDURE SHRINK(STRING:PACKED ARR
AY[1..20]OF CHAR);
 510    BEGIN;
 520     K:=20;
 530     WHILE STRING[K]=' ' DO
 540     K:=K-1;
 550    END;(*OF SHRINK*)
 560    PROCEDURE LIST$NAME;
 570    VAR J:INTEGER;
 580    BEGIN;
 590     FOR J:=1 TO MAX DO
 600      WITH MEMBER[J] DO BEGIN
 610       SHRINK(MEMBER[J].SURNAME);
 620       WRITE(SURNAME:K,',');
 630       SHRINK(MEMBER[J].FORENAME);
 640       WRITE(MEMBER[J].FORENAME:K,','
);
 650       WRITE(MEMBER[J].AGE:3,',');
 660       WRITELN(MEMBER[J].HANDICAP:2);
 670      END;
 680    END;(*OF LIST*)
 690    BEGIN;(*MAIN PROGRAM*)
 700     ENTER$NAME;
 710     SORT$NAME;
 720    PAGE;
 730     LIST$NAME;
 750    END.
```

Figure 6.1 Assigning complete records. This is done in lines 400 to 420 as part of the sort routine. In BASIC, you would have to assign each field of each record separately unless your record consisted of a long string.

Having looked at the main program, which consists of entry, followed by sorting and then display, take a look at the types and variables. The constant MAX is for the number of items that can be entered, and it has been kept to five for easy testing. Type PERSON is a record which consists of surname,

forename, age and handicap. In this record declaration, HANDICAP has been given as a range of integers, 0 to 24. If, incidentally, you wanted to cover the normal golf handicap range of +4 to −36, the easiest way to do so is to make the range 0 to 40, and get the actual golf handicap by taking 4-(HANDICAP). In the variable declaration section, MEMBER is the array of records, and TEMP has been defined as of PERSON, so that a record can be assigned to this variable. For a similar reason, STRING has been defined as another packed array of characters.

The procedure ENTER_NAME is straightforward, and uses the techniques that you should be familiar with by now. The procedure which is of most importance in this example is SORT_NAME, because this involves exchanging records. The plan of this procedure is a fairly standard Shell sort, and you will have to look this up in a reference book if you want more details on the sort routine itself. The important point is that a field of a record is compared with the same field of another record, and the records are interchanged if the fields are in the wrong order. In this example, the fields that are being compared are the handicaps, and line 390 is where the comparison takes place. Remember that variables such as HANDICAP are local in the WITH MEMBER[J] statements, so HANDICAP means nothing in this line, but items such as MEMBER[Z].HANDICAP *do* have a value. In this line, if the handicap of member Z is less than or equal to the handicap of member J, then lines 400 to 440 are carried out. In these lines, the record name TEMP is assigned to the value of MEMBER[Z]. MEMBER[Z] is then assigned with the value of MEMBER[J], and MEMBER[J] is assigned with the value of TEMPASCAL. This trio of assignments has the effect of swapping the values of record MEMBER[Z] and MEMBER[J]. The important point, once again, is that an *entire* record can be assigned in one statement.

When the records have been sorted in order of increasing handicap number, they are to be listed. Now when we list one item per line, as is usual, it doesn't matter that the fields are larger than the names. If, however, you wanted to shrink the field size so as to make room for something following the name, you have to know how many characters are in each name. This is done in the procedure SHRINK, which returns with a number (integer K) which measures the precise useful length of the string that has been entered. The idea of this is that it makes it possible to print the entire record on one line of the screen. If the 20-character lengths were retained, this would not be possible on a 40-character screen. SHRINK is the one of a number of procedures that you may want to use in your own programs, because Pascal is not quite so well-equipped with built-in instructions for string handling. As we go on, then, you will see several procedures introduced for dealing with strings, particularly for string display. These have been gathered together in Chapter 8.

Now suppose that we wanted to sort the list of names in some other way. In Figure 6.1, we chose to sort in order of ascending handicap number. Sorting by descending handicap is trivial; all you need to do is to reverse the < sign in line

390 into a > sign. The important change is to be able to sort 'by another field'. In plain language, this means sorting by surname order, or forename order, by age, or by any other feature. This is easy enough if you want the sort to be permanent. If, for example, you *always* wanted the program of Figure 6.1 to sort the records in order of surname, then all you have to do is to alter the program so as to use MEMBER[Z].SURNAME and so on in line 390. It's not quite so straightforward when sorting might be wanted by any of a number of fields, selected when the program runs. The obvious way to do this is to introduce a menu stage in the main program, and select each time the equivalent of line 390 when the program of Figure 6.1 runs. Figure 6.2 shows the amendments that can be made to allow a choice of three possible sort fields, in this case, surname, age or handicap. The main program has been arranged so that it will loop until the STOP key is pressed. This allows you to enter data and then test the action of the SELECT procedure.

Now look at the program of Figure 6.2 in detail. Two more variables have been declared. One is REPLY which will be used to accept a letter choice when you are asked to decide which field to use for sorting. The other is SWOP, which is a Boolean variable that will be used to decide whether to swop two records or not. Following the variables, there are several new procedures. Three of these are used to test fields. For each choice of field (age, handicap or surname), a procedure is used to test the appropriate fields of the records. The array numbers J and Z have to be passed to these procedures from the sorting procedure. In the course of a sort, only one of these procedures, the one that has been chosen by the SELECT procedure, will be used. In each case, if the test is true, then the Boolean variable SWOP is set to TRUE. It will be set to FALSE each time before the procedure is used.

The procedure SELECT carries out your choice of which field to sort by. The choices are listed, and you are asked to select by letter. The letter that you use is tested in line 360, and if you have not selected correctly, then a message is printed, the program waits for a key to be pressed, and the SELECT procedure is called again. This is an example of 'recursion', in which a procedure calls *itself*. When an acceptable letter is pressed, the procedure returns to the main program, with the value of a letter assigned to REPLY. This is then used in the SORT_NAME procedure, which starts in line 590. In the record exchanging part of this routine, the Boolean variable SWOP is set to false, and lines 720 to 750 carry out the selection of field. In these lines, then, the letter which has been coded as REPLY is used to select the correct TEST procedure, testing the correct field of each record. As a result of that test, SWOP will be either TRUE or FALSE. If SWOP is true, then the test in line 770 will cause the records to be exchanged. Note that you don't have to type

```
IF SWOP=TRUE...
```

only IF SWOP.., because this is the correct syntax in Pascal as it is in many versions of BASIC. The rest of the program then proceeds in the usual way.

This method of choosing which field to sort by can be used when the number

```
 10  PROGRAM P6N2;
 20  CONST MAX=5;
 30  MESG='PRESS ANY KEY...';
 40  TYPE PERSON=RECORD
 50         SURNAME,FORENAME:PACKED ARRAY[
1..20]OF CHAR;
 60         AGE:INTEGER;
 70         HANDICAP:0..24;
 80  END;
 90  VAR MEMBER:ARRAY[1..MAX]OF PERSON;
 100        TEMP:PERSON;
 110       J,K:INTEGER;
 120       STRING:PACKED ARRAY[1..20]OF CH
AR;
 130       REPLY:CHAR;
 140       SWOP:BOOLEAN;
 150  PROCEDURE TESTA(J,Z:INTEGER);
 160  BEGIN;
 170    IF MEMBER[Z].AGE<=MEMBER[J].AGE T
HEN SWOP:=TRUE;
 180  END;
 190  PROCEDURE TESTH(J,Z:INTEGER);
 200  BEGIN;
 210    IF MEMBER[Z].HANDICAP<=MEMBER[J].
HANDICAP THEN SWOP:=TRUE;
 220  END;
 230  PROCEDURE TESTS(J,Z:INTEGER);
 240  BEGIN;
 250    IF MEMBER[Z].SURNAME<=MEMBER[J].S
URNAME THEN SWOP:=TRUE;
 260  END;
 270  PROCEDURE SELECT;
 280  BEGIN;
 290    PAGE;
 300    WRITELN('PLEASE CHOOSE SORT FIELD
');
 310  WRITELN('A - AGE');
 320  WRITELN('H - HANDICAP');
 330  WRITELN('S - SURNAME');
 340  WRITE('YOUR CHOICE - ');
 350  READLN(REPLY);
 360  IF NOT(REPLY IN ['A','H','S']) THE
N
 370    BEGIN
```

```
  WRITELN('CHOICE IS A, H OR S ONL
Y');
  WRITELN(MESG);
  WHILE GETKEY=CHR(0) DO;
  SELECT;
  END;
END;
PROCEDURE ENTER$NAME;
VAR J:INTEGER;
BEGIN;
 FOR J:=1 TO MAX DO
  WITH MEMBER[J] DO BEGIN
   WRITE('SURNAME: ');
   READLN(SURNAME);
   WRITE('FORENAME: ');
   READLN(FORENAME);
   WRITE('AGE: ');
   READLN(AGE);
   WRITE('HANDICAP: ');
   READLN(HANDICAP);
  END;
END;(*OF ENTER$NAME*)
PROCEDURE SORT$NAME;
VAR I,Y,J,X,Z,IT:INTEGER;
BEGIN;
 Y:=1;
 WHILE Y<MAX DO Y:=2*Y;
  REPEAT
   Y:=TRUNC((Y-1)/2);
   IT:=MAX-Y;
   FOR I:=1 TO IT DO BEGIN;
    J:=I;
    REPEAT
     SWOP:=FALSE;
Z:=J+Y;
      CASE REPLY OF
      'A':TESTA(J,Z);
      'H':TESTH(J,Z);
      'S':TESTS(J,Z);
      END;
     IF SWOP THEN BEGIN
      TEMP:=MEMBER[Z];
      MEMBER[Z]:=MEMBER[J];
      MEMBER[J]:=TEMP;
```

```
 810        J:=J-Y;
 820        END
 830      ELSE J:=0;
 840       UNTIL J<=0;
 850    END;
 860  UNTIL Y=1;
 870  END;(*OF SORT*)
 880  PROCEDURE SHRINK(STRING:PACKED ARR
AY[1..20]OF CHAR);
 890  BEGIN;
 900   K:=20;
 910   WHILE STRING[K]=' ' DO
 920   K:=K-1;
 930  END;(*OF SHRINK*)
 940  PROCEDURE LIST$NAME;
 950  VAR J:INTEGER;
 960  BEGIN;
 970   FOR J:=1 TO MAX DO
 980    WITH MEMBER[J] DO BEGIN
 990     SHRINK(MEMBER[J].SURNAME);
1000      WRITE(SURNAME:K,',');
1010      SHRINK(MEMBER[J].FORENAME);
1020      WRITE(MEMBER[J].FORENAME:K,','
);
1030      WRITE(MEMBER[J].AGE:3,',');
1040      WRITELN(MEMBER[J].HANDICAP:2);
1050     END;
1060  END;(*OF LIST*)
1070  BEGIN;(*MAIN PROGRAM*)
1080   ENTER$NAME;
1090    REPEAT
1100     SELECT;
1110     SORT$NAME;
1120     PAGE;
1130     LIST$NAME;
1140     WRITELN(MESG);
1150     WHILE GETKEY=CHR(0)DO;
1160    UNTIL FALSE;
1180  END.
```

Figure 6.2 A program which allows a choice of sort fields. This has been developed from the previous program.

of records is comparatively small, but causes problems when a large number of records are used. The problem is one of sorting time. The CASE selection line has been placed in a part of the sorting loop which is repeated very many times during a sort, and as a result, has a large effect on the total time that is taken. Because Pascal is a compiled language, the effect is not so serious as it would be in BASIC, but it can make the sort action irritatingly slow. If you need to exercise a choice of sort field for a list of records which consists of a large number of items, all held in the memory, then a faster option is to have as many sort procedures as you want sort fields, and to select the complete procedure. You might, for example, have procedures SORTNAME, SORT_HCAP and SORT_AGE which were selected by the value of REPLY. Each of these procedures would be identical apart from the test line IF MEMBER[Z].FIELD<=MEMBER[J]>FIELD in each one. This method requires more code, but runs much faster. The reason is that the choice is not having to be enforced each time two items are being compared. The choice is made once and used to select a procedure which can then run unencumbered. As usual, when you design a program, you can design it to run fast, or you can design it to be comparatively short, but you can't normally have both!

RECORD NESTS AND CHOICES

So far, each record that we have illustrated has consisted of items that are simple variables. We can, however, use records which consist partly or completely of other records! Records which are a part of another record are called 'nested' records, and typically they are used to hold details of an entry. We might, for example, have an entry called BIRTH which would require the details of day, month and year of birth. This could be provided by making BIRTH a record in itself, with DAY, MONTH and YEAR items of that record. Figure 6.3 shows how this provision for nested records can be used. The main record, as before, is called PERSON, but it now contains the sub-records MEMNAME and BIRTH. MEMNAME is of type NAME, and BIRTH is of type DOB, both of which must be defined as records *before* the main record can be defined. NAME is defined as consisting of SURNAME and FORENAME, both packed arrays of CHAR, and DOB consists of DAY, MONTH and YEAR, all integers with defined ranges. These ranges would, in a working program, be checked each time an item was entered.

Because the records are nested, we need nested loops to read items into the records, and similar nested loops to write the items. For the sake of simplicity, all stages except entry and printing have been omitted from this program. The entry procedure starts in line 190, and the important feature here is that the outer record loop is the one which uses MEMBER[J]. Within this loop, there are two sub-loops. One uses MEMNAME, the name of the sub-record that consists of surname and forename. The other sub-loop is BIRTH, consisting of

```
 10  PROGRAM P6N3;
 20  CONST MAX=2;
 30  TYPE NAME=RECORD
 40       SURNAME,FORENAME:PACKED ARRAY[1
..20]OF CHAR;
 50  END;
 60        DOB=RECORD
 70        DAY:1..31
 80        MONTH:1..12;
 90        YEAR:1900..2000;
 100  END;
 110       PERSON=RECORD
 120       MEMNAME:NAME;
 130       BIRTH:DOB;
 140       PHONE:PACKED ARRAY[1..14]OF CH
AR;
 150  END;
 160  VAR MEMBER:ARRAY[1..MAX]OF PERSON;
 170      J,K:INTEGER;
 180      STRING:PACKED ARRAY[1..20]OF CH
AR;
 190  PROCEDURE ENTER$NAME;
 200  VAR J:INTEGER;
 210  BEGIN;
 220   FOR J:=1 TO MAX DO
 230    WITH MEMBER[J] DO BEGIN
 240      WITH MEMNAME DO BEGIN;
 250       WRITE('SURNAME: ');
 260       READLN(SURNAME);
 270       WRITE('FORENAME: ');
 280       READLN(FORENAME);
 290      END;
 300      WITH BIRTH DO BEGIN
 310       WRITE('DAY OF BIRTH 1-31 ');
 320       READLN(DAY);
 330       WRITE('MONTH OF BIRTH 1-31 ')
;
 340       READLN(MONTH);
 350       WRITE('YEAR OF BIRTH 1900-200
0 ');
 360       READLN(YEAR);
 370      END;
 380     WRITE('PHONE NUMBER ');
 390     READLN(PHONE);
```

```
400    END;
410  END;(*OF ENTER$NAME*)
420  PROCEDURE LIST$NAME;
430  VAR J:INTEGER;
440  BEGIN;
450   FOR J:=1 TO MAX DO
460    WITH MEMBER[J] DO BEGIN
470     WRITE('SURNAME: ');
480     WRITELN(MEMBER[J].MEMNAME.SURNA
ME);
490     WRITE('FORENAME: ');
500     WRITELN(MEMBER[J].MEMNAME.FOREN
AME);
510     WRITE('DATE OF BIRTH: ');
520     WRITE(MEMBER[J].BIRTH.DAY:2,'
');
530     WRITE(MEMBER[J].BIRTH.MONTH:2,'
  ');
540     WRITELN(MEMBER[J].BIRTH.YEAR:4)
;
550     WRITE('PHONE: ');
560     WRITELN(PHONE);
570     WRITELN
580    END;
590  END;(*OF LIST*)
600  BEGIN;(*MAIN PROGRAM*)
610   ENTER$NAME;
620   PAGE;
630   LIST$NAME;
650  END.
```

Figure 6.3 Using nested records, in which a record can contain fields which are themselves records. This allows you to define a record, and then decide what detail you need.

DAY, MONTH and YEAR. Following the sub-loops or inner loops, the phone number is obtained. Note that this is put into string form. A telephone number is most unlikely to be expressible as an integer, and because it may contain dashes (as in 0999–1123–212), it cannot be expressed as a real number either.

The full effect of using nested records is illustrated in the LIST_NAME procedure, line 420 onwards. The loop construction is just as it was before, but the items are more detailed. An item such as MEMBER[J].MEMNAME.SURNAME is used to show that this is the surname field of record MEMNAME, which is itself a field of record MEMBER[J]. The sub-field extension is specified by using full-stops between the items. Note how the date of birth items have

been fielded so as to make the date look better on the screen. This example shows the most awkward type of sub-record identification, in which there are several sub-fields of one main field. Where one record is nested entirely inside one other, we can avoid using WITH more than once by using a line such as

WITH MEMBER[J].MEMNAME DO BEGIN...

with items such as SURNAME and FORENAME. This works only when *all* the items can be input into the MEMNAME record.

Sometimes a choice can be made about a record when the record is being made. If, for example, you were creating a file of names and addresses you might want to distinguish male from female names so that for female names you could enter maiden name in addition to married name (if such distinctions are allowed in the brave new world). It is possible to construct the RECORD in such a way that if an answer is M, then name is asked, but if the answer is F, then name and maiden name is asked. A record which allows such variations is called a 'variant record'. The variation is carried out by constructing a CASE statement within the record definition itself.

Take a look, by way of example, at the program in Figure 6.4. This does not,

```
 10  PROGRAM P6N4;
 20  CONST MAX=2;
 30  TYPE ACTION=(ADD,REVISE,DELETE);
 40       NAME=RECORD
 50       NUMBER:INTEGER;
 60       CASE CHOICE:ACTION OF
 70          ADD:(SURNAME,FORENAME:PACKE
D ARRAY[1..20]OF CHAR
 80          AGE,HANDICAP:INTEGER);
 90       REVISE:(HANDICAP,AGE:INTEGER);
 100         DELETE:();
 110    END;
 120  VAR PERSON:NAME;
 130           J:INTEGER;
 140  PROCEDURE ADDIT;
 150   BEGIN;
 160   WITH PERSON DO BEGIN;
 170    CHOICE:=ADD;
 180    WRITE('SURNAME: ');
 190     READLN(SURNAME);
 200     WRITE('FORENAME: ');
 210     READLN(FORENAME);
 220     WRITE('HANDICAP: ');
 230     READLN(HANDICAP);
 240     WRITE('AGE: ');
 250     READLN(AGE);
```

```
260     END;
270   END;
280   PROCEDURE REVISIT;
290    BEGIN;
300    WITH PERSON DO BEGIN;
310     CHOICE:=REVISE;
320     WRITELN('HANDICAP: ');
330     READLN(HANDICAP);
340     WRITELN('AGE: ');
350     READLN(AGE);
360    END;
370   END;
380   PROCEDURE DELETIT;
390    BEGIN
400     WITH PERSON DO BEGIN
410      CHOICE:=DELETE;
420      SURNAME:='
;
430      FORENAME:='
';
440      HANDICAP:=0;
450      AGE:=0;
460      END;
470      END;
480   PROCEDURE CHOOSE;
490   VAR REPLY:CHAR;
500    BEGIN;
510     WRITELN('DO YOU WANT TO- ');
520     WRITELN('ADD A NEW NAME,');
530     WRITELN('REVISE AN ENTRY,');
540     WRITELN('DELETE AN ENTRY');
550     WRITELN('(TYPE A,R OR D)');
560     READLN(REPLY);
570     CASE REPLY OF
580      'A':ADDIT;
590      'R':REVISIT;
600      'D':DELETIT;
610     END;
620   END;
630   BEGIN(*MAIN*);
640    CHOOSE;
660   END.
```

Figure 6.4 Illustrating variant records, in which a choice is made in the record definition itself.

in itself, do much except to illustrate variant records in action, but it can be expanded to provide you with something more useful. In line 30 the word ACTION is defined to be of ADD, REVISE or DELETE, and in line 40 the definition of NAME starts. This defines an integer NUMBER (not used in the example), and then a choice of items which depends on whether we are adding, revising or deleting entries. The variable PERSON is then defined as being a record of NAME, and J (not used) as an integer. The important point here is the CASE statement which is *within* the record definition. An identifier CHOICE is used, and the syntax is to follow CASE with the identifier, a colon, and then the identifier type (which is ACTION). In the lines of choice that follow, each choice item is followed by a colon and then a list of its variables within brackets. The definition may occupy more than one line, as in lines 70 and 80, in which case there should be no semicolon between lines. The DELETE option has no entries (nothing is entered when an item is deleted), but the brackets must be included.

The main program (lines 640–660) asks only for the procedure CHOOSE. In this procedure, you are asked what course of action you want, and the single character reply is used to select one of three procedures. In these procedures, entries can be made to a record, items amended, or a record deleted—and that's it! The important part is the variation of the record. It's not necessarily a feature that you will need to use very much, but it's there. You are allowed only one variant in a record—but the variant can be nested. In other words, you can have a variant which is part of another variant. If your planning is up to it, you can cope with quite complicated requirements in this way.

POINTERS

A pointer is a number variable, and the reason for its name is that it 'points' to something. For example, suppose that you have the character C stored in the memory of the Commodore 64. What this actually means is that one of the memory cells is storing the number which is the ASCII code for C, the number 67. Now memory for a computer is organised so that each unit (or *byte*) is numbered, and we might know that the number of the byte which held our C was 26 000. This number of 26 000 is the number which is the pointer for C. We could, if we liked, store the number 26 000 somewhere so as to make it possible for the computer to find where C was stored, and this is precisely what the action of a pointer is. It would be rather a waste to store a pointer for every character, but we don't need to. All we need to do is to store a pointer to the *start* of any variable. Once we know where the start is, we can locate it and read the required number of bytes of data. This is something that is used a lot in assembly language programming, but seldom occurs in BASIC. A few machines, notably the MSX machines, have a BASIC function called VARPTR which comes back with a number that tells you whereabouts in the memory a

variable is stored. In BASIC, however, there is little use for this action, and not many programmers make use of it, or are even aware of it. In Pascal, however, pointers are a way of storing variables which can be very useful indeed.

A pointer in Pascal is a variable quantity which is the address of another variable. What makes the pointer valuable is that if the pointer is declared, the compiler does not need to have the other variable declared. For example, if the pointer to a real number is known, and is variable X, then the name of the real number does not have to appear earlier in the program. It is even possible to allocate memory space for a quantity when the program is running. Normally, this space is allocated when you compile.

A pointer reserves space; what you put into the space later is your own business. In addition, the pointer is a number, but what it points to can be any type of variable, simple (such as another integer) or structured (like a record). We can then juggle with the pointers rather than with the variables themselves. Most important of all, when pointers are used rather than variable names, the memory is used in a different way. Storage is allocated in a part of the memory which is called the 'heap', often located low in memory. This allocation is not fixed. When you declare an array of one hundred strings, each of twenty characters, you have automatically tied up (at least) one thousand bytes of memory. This will be tied up whether you use it or not. If you allocate by way of pointers, you will allocate only what you need, and when you need it. This does not, however, mean that using pointers will allow you to squeeze longer programs into your computer, because you may need to use a lot of memory in managing the pointer system. Like all programming, what you gain on the swinging pointers, you lose on the management roundabouts.

Take a look at the example in Figure 6.5. This illustrates in a reasonably simple way the allocation and use of a pointer. A data type STRING is declared as a packed array of twenty characters, and then variable T is declared as a

```
 10  PROGRAM P6N5;
 20  TYPE STRING=PACKED ARRAY[1..20] OF
CHAR;
 30  VAR T:^STRING;
 40  BEGIN;
 50  NEW(T);
 60  T^:='THIS IS A STRING     ';
 70  WRITELN(T^);
 80  T:=NIL;
 90  WRITELN(T^);
 110  END.
```

Figure 6.5 Allocating and using a pointer. This shows the pointer used as if it were a variable name.

pointer to STRING. Note how this is typed, as VAR T:^STRING. This quantity T which is the pointer is not an ordinary variable type. Its value cannot be printed or input; it cannot be changed in any way by you. All you can do with it is to allocate it to a quantity and de-allocate it. In the program itself, the instruction NEW(T) is the allocation instruction. It causes T to carry the pointer to STRING. We can then allocate something to this string. Note that we have no variable name for the quantity, only a pointer. Line 60 then declares that T points to the phrase THIS IS A STRING (padded out to twenty characters). We can prove this by the WRITE instruction in line 70, and then T can be pointed to address 0000 by using T:=NIL. This is the only way of making T equal to a known number. When line 90 is executed, you will see a 'nonsense' line, the characters which do not correspond to the first few bytes of memory. If you replace line 80 by the instruction DISPOSE(T), you will see when you compile and run again, that the string is still printed. DISPOSE does not mean that the stored characters are removed, simply that the pointer can be re-used for other purposes. Since in this case we haven't used it for anything else, it still allows us to find the string.

Now for another step forward. There is no reason why we should not have an array of pointer variables, each one of which points to something. Figure 6.6 shows how this could be arranged, with STRING declared as an array of CHAR as before, and type T declared as pointer to type STRING. When the variables are declared, TP is an array with five values which is of pointers of type T, and J is an integer to use in counting loops. Now in the loop of lines 70 to 110, each element of the pointer array is allocated by using NEW, and is then assigned

```
 10  PROGRAM P6N6;
 20  TYPE STRING=PACKED ARRAY[1..20] OF
CHAR;
 30       T=^STRING;
 40  VAR TP:ARRAY[1..5]OF T;
 50       J:INTEGER;
 60  BEGIN;
 70  FOR J:=1 TO 5 DO BEGIN
 80    NEW(TP[J]);
 90    WRITELN('STRING? ');
100    READLN(TP[J]^);
110  END;
120   FOR J:=1 TO 5 DO BEGIN;
130    WRITELN(TP[J]^);
140   END;
160  END.
```

Figure 6.6 Using an array of pointers to carry out actions on string arrays.

with a string to point to by the READLN statement. The loop in lines 120 to 140 shows that the assignments have been carried out. Of course, there's no particular point in using an array of pointers when we could just as easily have an array of strings, but the important point is that it demonstrates that we can use an array of pointers. We seldom, in fact, use an array of pointers because there is a much more useful method of using pointers, which is as part of records. The trouble with this is that it's notoriously difficult to explain!

LINKED LISTS

Suppose that you defined a record which consisted of a real number (it makes a change from characters) and a pointer. Now the most important feature of a pointer is that it can be made to point to something that may be anywhere in the memory of the computer. Because this is possible, we can make the pointer in the record point to the next record, even if this means a record which is not the next one that you enter, or even the previous one. If you make up a set of records like this, you don't need an array. Each record contains a pointer to the next record so that if you can locate the first record, you can get to any other, swinging like Tarzan on ropes of pointers from record to record. Figure 6.7 shows in diagram form what this is all about. A sequence like this is called a 'linked list'.

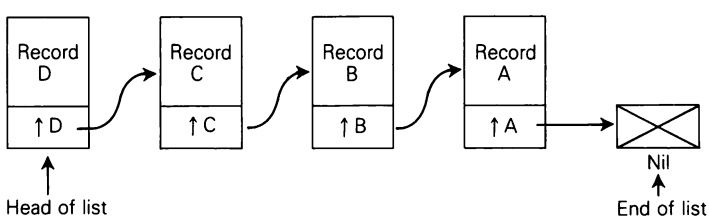

Figure 6.7 What a linked list consists of. Each item in the list, which would be a record, carries a pointer to another item. This would normally point to the 'next' record, which is the one nearer the end of the list.

What advantages would such a structure have? Well for one thing, it becomes very easy to 'delete' a record. All you need to do is to alter the pointer that points to the item and make it point to the next one instead (Figure 6.8). Having coped with that idea, how would you reverse the order of two records? Figure 6.9 shows the principle in diagram form, requiring three pointers to be changed. What we have to do now is to see how some of this paper talk can be transferred into a working program. Figure 6.10 shows a typical example which is deceptively simple. We define POINT as a pointer type to the record CASH. CASH is then defined as a record which contains DAILY, a real number, and

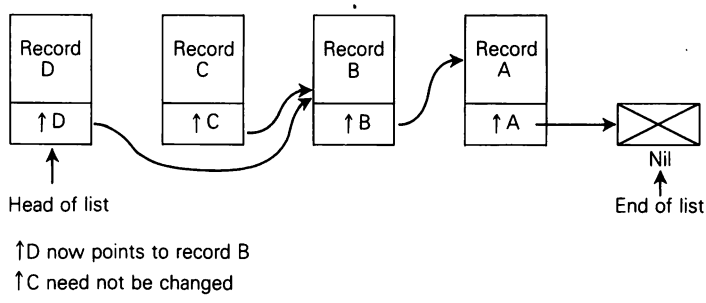

Figure 6.8 How a record is deleted from a linked list. If there is no pointer to an item, then it doesn't exist!

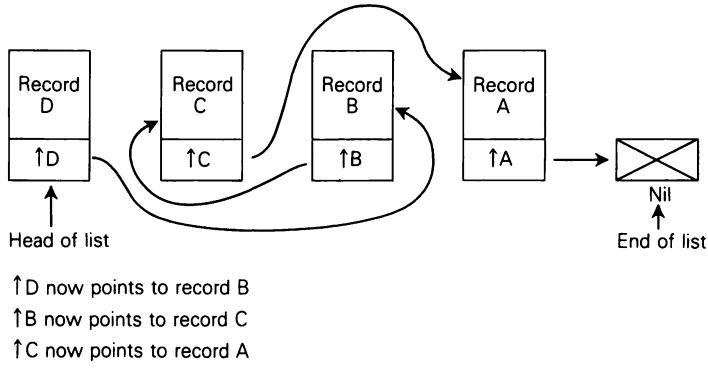

Figure 6.9 How the order of two records in a linked list can be changed.

NEXT, a pointer. The variables FIRST and P are defined as pointers, and J as an integer, and the program itself starts in line 90. This starts by making FIRST point to NIL. There are no values to point to yet, so the pointer points nowhere. A loop is then set up. Unlike an array, there is no preset limit to size here, and entry can continue until you enter a figure of zero, or until the memory is full. In a real program, the data would be stored on disk, and it would take quite some time to fill that!

Now what happens in the loop is vital to the way that the list is constructed, and you need to follow it very carefully. I think it's a lot easier if we can think of numbers for the pointers, and so I'll assume that the pointer NIL is zero (true), and that the other pointers will be 100, 200, 300 and so on (undoubtedly false, but it helps you to understand it). Let's take a walk through the first loop round. The NEW(P) in line 120 will allocate a pointer, perhaps 100. This will now be used to store the cash amount, P^.DAILY. The pointer quantity NEXT for this record is now made equal to FIRST, which is NIL. This is the way of signalling that there is no following record. FIRST is then made equal to P, assumed to be 100.

```
10  PROGRAM P6N10;
20  TYPE POINT=^CASH;
30  CASH=RECORD
40    DAILY:REAL;
50    NEXT:POINT;
60  END;
70  VAR FIRST,P:POINT;
80              J:INTEGER;
90  BEGIN;
100  FIRST:=NIL;
110  REPEAT
120    NEW(P);
130    WRITELN('TODAYS CASH:- ');
140    READ(P^.DAILY);
150    P^.NEXT:=FIRST;
160    FIRST:=P;
170  UNTIL P^.DAILY=0;
180  PAGE;
190  WHILE GETKEY=CHR(0) DO;
200  J:=1;
210  P:=FIRST;
220  WHILE P<>NIL DO
230  BEGIN;
240    WRITELN(J,' - ',P^.DAILY:1:2);
250    P:=P^.NEXT;
260    J:=J+1;
270   END;
290  END.
```

Figure 6.10 Entering items into a linked list until zero is entered. The list can then be listed on the screen.

So much for the first loop. What happens in the second? We pick a new P allocation, assumed 200 this time. Once again, we store a cash quantity, and the NEXT pointer is this time made equal to 100, the pointer to the previous entry. FIRST is now made equal to P, which is 200. If you look at these steps in diagrammatic form (Figure 6.11) you can see that the list is not growing in the way that you might expect. The 'top' of the list is the item that we entered last of all. Its pointer always points to the next one down, the previous item. The list ends with the one that points to FIRST. If any more proof is needed, take a look at what lines 200 to 270 print out. Enter items like 11.1, 22.2 and so on that you can recognise. When you see the listing it will show 1–0.00, which is the zero entry that you used to close the list. After that your other values follow in reverse order of entry, with the most recently entered item as item 2 and the

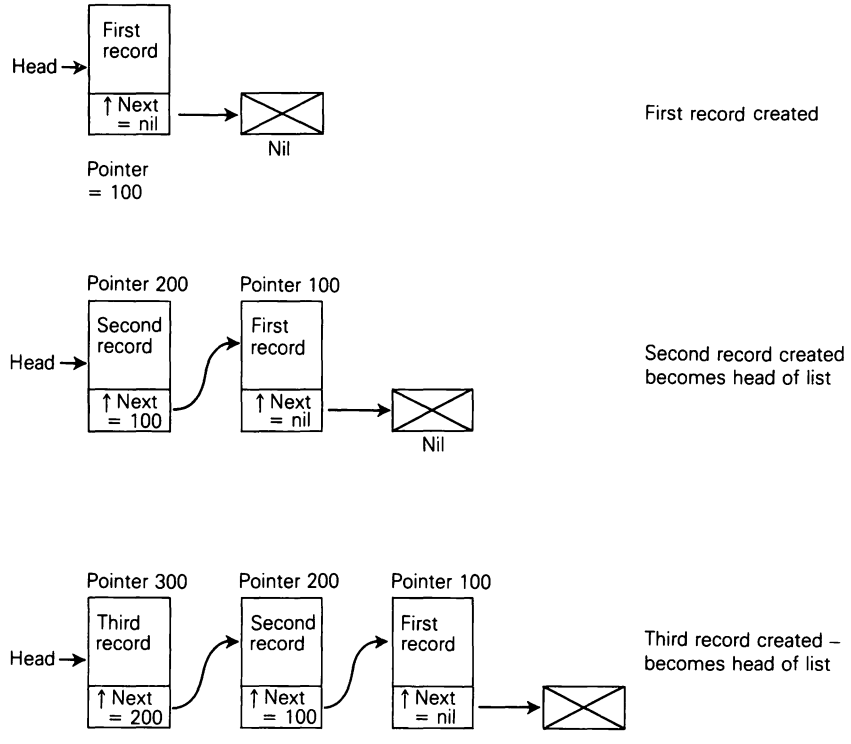

Figure 6.11 How the list grows, with each new item becoming the head of the list.

first item that you entered at the end. The word NEXT can be rather confusing in program examples like this. It certainly means the next item in the list, but it's the next one back simply because there isn't a next one forward until you create one!

Could we, as a matter of interest, make a list in a different way? When you think about it, there's no obvious method. You can't direct a pointer to the next item, because until you create one it doesn't exist. Once you carry out a NEW(P) to get the next pointer, you have lost all trace of the previous one. Though it is possible to construct what are called 'double-linked' lists, with a pointer in each direction, programming of that sort is beyond the scope of this book, and if you need to use it, you probably need a lot more memory for your program than you'll find in the C-64! What is more important at this point, then, is learning how to operate on the lists so that you can search through a list.

Figure 6.12 shows how you can search through a list for an item. You need to know, of course, what you are looking for, whether it's the item with a value of 6.66 or the first one whose value is less than 9 or whatever criterion you adopt. You need not work, of course, with REAL number values in the records, as long

```
 10  PROGRAM P6N11;
 20  TYPE POINT = ^CUSTOMER;
 30       CUSTOMER=RECORD
 40       NUMBER:INTEGER;
 50       SURNAME:PACKED ARRAY[1..20]OF
CHAR;
 60       BALANCE:REAL;
 70       NEXT:POINT;
 80       END;
 90  VAR FIRST,P:POINT;
 100           J:INTEGER;
 110         LOST:BOOLEAN;
 120  BEGIN;
 130  FIRST:=NIL;
 140  REPEAT;
 150    NEW(P);
 160    WRITE('NUMBER: ');
 170    READLN(P^.NUMBER);
 180    WRITE('SURNAME: ');
 190    READLN(P^.SURNAME);
 200    WRITE('BALANCE: ');
 210    READLN(P^.BALANCE);
 220    P^.NEXT:=FIRST;
 230    FIRST:=P;
 240    UNTIL P^.NUMBER=0;
 250  PAGE;
 260  REPEAT;
 270   WRITELN('PRESS ANY KEY...');
 280   WHILE GETKEY=CHR(0) DO;
 290   WRITELN('NUMBER, PLEASE');
 300   READLN(J);
 310   LOST:=TRUE;
 320   P:=FIRST;
 330   WHILE LOST DO
 340    IF P^.NUMBER=J THEN
 350    BEGIN;
 360    WRITELN('NUMBER ',J);
 370    WRITELN('NAME ',P^.SURNAME);
 380    WRITELN('BALANCE ',P^.BALANCE:1:
2);
 390    LOST:=FALSE;
 400    END
 410   ELSE IF P^.NEXT=NIL THEN
 420      BEGIN;
```

```
430         LOST:=FALSE;
440         WRITELN('NOT FOUND');
450         END
460       ELSE P:=P^.NEXT
470     UNTIL FALSE;
490     END.
```

Figure 6.12 How to search a linked list for an item.

as each record contains some data that you want to use along with a pointer to the next record. In this example, each record consists of an identification number, a name, a cash balance, and the usual pointer. The aim is to find the details for a given identification number, but you could, of course, easily adapt the program to find a name, or any names beginning with a given letter, or names with a balance exceeding 100, or any other conditions you like to use. The item-finding part has been put into a loop so that you can try the options of a number that you know is genuine and one that you know is not in the list. You can escape in the usual way by pressing the STOP key when you see the PRESS ANY KEY message.

The records are set up in the usual way in lines 120 to 240, and you know already how this is done. The search portion of the program is in lines 260 to 470. We start with the PRESS ANY KEY message and the GETKEY step, then you are asked for a number, the identity number. This is assigned to integer variable J, and the Boolean variable LOST is set to TRUE. The pointer P is set to the value FIRST—this is sometimes referred to as the 'head' of the list. A loop is then performed for as long as LOST is TRUE. Lines 340–390 deal with the case of the pointer pointing to the correct number. The number and the other details of the record are then printed out. If the record that has just been read carries a pointer to another record, then lines 410–450 are not used, and P is re-assigned to the next pointer value in line 460. This repeats until one of two events happens. One is that you find the item that you are looking for, and LOST is set to FALSE. The other possibility is that you reach the end of the list, and so P^.NEXT is NIL. This also sets LOST to FALSE to prevent any further looping, and also prints a message to the effect that the item is not found. That's all you need, and the search is a very fast one, even if quite a number of records are in use.

From this point on, the topic of linked lists starts to get complicated, and you will need to consult specialised texts if you want to see how to insert or delete items. It's advisable to make sure that you have a really firm grip on Pascal programming before you attempt such work, and that's why I shall not pursue the topic in this chapter. The next topics, in fact, are a long way removed from database programs—how to harness the sound and graphics instructions of the Commodore 64 in Oxford Pascal.

7

GRAPHICS AND SOUND

The BASIC of the Commodore 64 copes reasonably well with the low-resolution graphics display, but has no commands that refer to the high-resolution graphics. The Commodore 64 manual, in fact, doesn't mention the topic, and users of the machine might not even know that high-resolution graphics were possible unless they saw a display on another machine. Like the sound effects, the high-resolution graphics are reached only through the use of machine code, or by using POKE instructions from BASIC. Oxford Pascal remedies this situation by allowing you to switch to a high-resolution display, and to construct patterns by means of a few useful instructions. This allows you access to high-resolution graphics without all of the complications of learning machine code, or of knowing the correct memory addresses that have to be poked from a BASIC program.

The high-resolution screen display is switched on by using HIRES(1), and off again by using HIRES(0). These, like the other graphics and sound instructions in Oxford Pascal are no part of standard Pascal, because standard Pascal is designed to be used with any machine, and graphics instructions must necessarily vary from one machine to another. Once in high-resolution mode, you can control colours of background (paper) and plotting (ink) by making use of PAPER and INK, each of which can be followed, within brackets, by one of the Commodore 64 colour numbers 0 to 15. Figure 7.1 shows you the sort of colour effects that you can expect. In this program, the screen is switched to HIRES(1), and PAPER and INK colours are specified. The GETKEY step allows the display to remain in high-resolution mode until you press any key. When you run this one, you will see the screen background change, but you will also see a jagged pattern of black stripes on the screen. This looks very discouraging, and you have to remove it by using two further commands. These are based on the PLOT statement, which is a multipurpose type of command that can carry out different actions depending on what numbers follow PLOT, within brackets. Of

```
10  PROGRAM P7N1;
20  BEGIN;
30  HIRES(1);
40  PAPER(7);
50  INK(2);
60  WHILE GETKEY=CHR(0) DO;
70  HIRES(0);
90  END.
```

Figure 7.1 Switching in **HIRES**. The effect is not useful until you clear the screen.

these numbers, the first one is the one that decides what PLOT will do. If this first number is 0, then the screen will be cleared to the paper colour. If this first number is 1, then all of the high-resolution points on the screen are cleared. To clear the screen completely, then, we need to use a PLOT(0...) and a PLOT(1...). Since the PLOT instructions need five numbers, separated by commas, we can use zeros for the other numbers for the moment. The modified program is shown in Figure 7.2, and the result of this one is a lot more pleasing! If you don't use HIRES(0) at the end of a program, the screen will remain in high-resolution mode, and you will not see any text appear when you press keys. You can escape from this condition by typing (very carefully) KILL (then RETURN). You will not see the word KILL appear on the screen, which is why you need to be careful.

```
10  PROGRAM P7N2;
20  BEGIN;
30  HIRES(1);
40  PAPER(7);
50  INK(2);
60  PLOT(0,0,0,0,0);
70  PLOT(1,0,0,0,0);
80  WHILE GETKEY=CHR(0) DO;
90  HIRES(0);
110  END.
```

Figure 7.2 Using two **PLOT** statements to clear the high-resolution screen.

The full syntax of the PLOT instruction is as follows.

PLOT(A,X1,Y1,X2,Y2);

A can take numbers 0 to 5:

A=0 Clear screen to colour selected by PAPER.

A=1 Clear all INK points (clear screen command).

A=2 Plot a line from X1,Y1 to X2,Y2.
A=3 Unplot or clear line from X1,Y1 to X2,Y2.
A=4 Fill a closed area round X,Y with INK colour. X1=X2=X and Y1=Y2=Y.
A=5 Clear a filled area round X,Y.

The numbers 2 to 5 immediately following the PLOT(part are the important ones from the point of view of creating patterns. They require four more numbers, which are position co-ordinates X and Y for the start and end of a line. The limits of these numbers are 0 to 255 for X and 0 to 200 for Y. If you use larger numbers you will not get an error message, but the plot will not extend further than would be given by these limits. The limit of 255 on X means that about 20 per cent of the high-resolution screen area on the right-hand side is not used. Another point which is not evident from the manual is that high-resolution plotting, even in a fast language like Pascal, is never really fast, certainly not as fast as can be achieved with direct machine-code control over the high-resolution screen. In addition, you will see irritating flickering effects which are due to lack of synchronisation between the graphics actions and the scanning of the screen.

This is best illustrated by an example, and Figure 7.3 shows a program which creates a fan-like pattern on the screen. Note that because you cannot use any STEP instruction in Pascal, the FOR loop uses numbers which are multiplied by 4 to get the X co-ordinate number for the display. In addition, the program uses a range of numbers for X which are well beyond the limit of 255. This does not provide any display beyond the limit of the screen, but it ensures that the height

```
10  PROGRAM P7N4;
20  VAR J,K:INTEGER;
30  BEGIN;
40  HIRES(1);
50  PAPER(7);
60  INK(2);
70  PLOT(0,0,0,0,0);
80  PLOT(1,0,0,0,0);
90  K:=200;
100  FOR J:=0 TO 300 DO
110   BEGIN;
120    PLOT(2,0,0,4*J,K);
130   END;
140  WHILE GETKEY=CHR(0) DO;
150  HIRES(0);
170  END.
```

Figure 7.3 A program that uses **PLOT** to create a fan pattern.

which is reached by the value of K is less on each successive run with an X value of more than 255. The pattern also shows *moiré* (pronounced mwarrey) *effects*, the odd curves and colours that are the result of interacting patterns. One of the patterns is the pattern on the screen itself; the other is the pattern of holes in the shadowmask of the colour VDU tube. The pattern builds up at a reasonable speed at first, but as the lines start to overlap, the changes become less easy to see. You will also notice that the occasional flicker over the picture reveals the words that will appear on the screen after the program has been completed. Most of this flickering affects the bottom part of the area, and if we can use this for an unimportant part of the display, all the better.

Figure 7.4 demonstrates how a pattern can be plotted and 'unplotted'. The program contains three procedures, SQUARE, UNSQ and WAIT. The SQUARE routine draws a square by using four line plot instructions. The unsquare procedure, UNSQ, wipes out each of the same lines that SQUARE has drawn. The WAIT procedure puts in a time delay between the other two procedures. The number limit that is used in this routine has to be chosen carefully. The REPEAT action is surprisingly slow, and a limit of, for example, 1000 will result in agonisingly slow drawing and wiping. This is not particularly important for the C-64, because if you want to work with animated sprites, the facilities are present to some extent in BASIC. The POKE command of BASIC can be used from Pascal provided that you use the correct syntax, and so sprites can be implemented (along with anything else that can be implemented by a POKE action) from Pascal as easily (or with as much difficulty) as from BASIC.

THE FILL INSTRUCTION

The PLOT instructions that use codes 4 and 5 implement colour filling of any closed area, provided that the co-ordinates that are used in the instruction are the co-ordinates for a point which is within the area. In an instruction of this type, which calls for only one pair of co-ordinates, you should make the two sets identical, because the PLOT instruction demands that one mode number and four co-ordinate numbers are always entered. It is important to stress that *the area must be closed.* If it is not, the filling routine cannot operate correctly, and you will find colour 'leaking' from where you want it and slowly spreading over the rest of the screen. Figure 7.5 shows this instruction in action. The PLOT lines 110 to 140 draw a square, and line 150 carries out the filling action. The two sets of co-ordinates are for a point within the square. Any point that is within the boundary will do, but you should not use a point which is on the boundary itself, and you must be certain that the square is closed. As before, the GETKEY line holds the program in high-resolution mode until you press any key. If you forget this step, and have no HIRES(0) at the end of the program, then the screen will remain in high-resolution mode, and no text will appear. If you find yourself stuck like this, remember that you don't need to

```
10  PROGRAM P7N5;
20  VAR X,Y:INTEGER;
30  PROCEDURE SQUARE(X:INTEGER);
40  BEGIN;
50   PLOT(2,X,Y,X+8,Y);
60   PLOT(2,X+8,Y,X+8,Y-8);
70   PLOT(2,X+8,Y-8,X,Y-8);
80   PLOT(2,X,Y-8,X,Y);
90  END;
100  PROCEDURE UNSQ(X:INTEGER);
110   BEGIN;
120    PLOT(3,X,Y,X+8,Y);
130    PLOT(3,X+8,Y,X+8,Y-8);
140    PLOT(3,X+8,Y-8,X,Y-8);
150    PLOT(3,X,Y-8,X,Y);
160   END;
170  PROCEDURE WAIT;
180  VAR J:INTEGER;
190  BEGIN;
200  J:=0;
210  REPEAT
220   J:=J+1;
230   UNTIL J>20;
240  END;
250  BEGIN; (*MAIN PROGRAM*)
260  HIRES(1);
270  PAPER(7);
280  INK(2);
290  PLOT(0,0,0,0,0);
300  PLOT(1,0,0,0,0);
310  Y:=104;
320  FOR X:= 0 TO 50 DO
330   BEGIN;
340    SQUARE(4*X);
350    WAIT;
360    UNSQ(4*X);
370   END;
380  WHILE GETKEY=CHR(0) DO;
390  HIRES(0);
410  END.
```

Figure 7.4 Plotting and clearing with **PLOT** numbers 2 and 3.

```
10  PROGRAM P7N6;
20  VAR X,Y:INTEGER;
30  BEGIN;(*MAIN PROGRAM*)
40  HIRES(1);
50  PAPER(7);
60  INK(2);
70  PLOT(0,0,0,0,0);
80  PLOT(1,0,0,0,0);
90  X:=100;
100  Y:=75;
110  PLOT(2,X,Y,X,Y+50);
120  PLOT(2,X,Y+50,X+50,Y+50);
130  PLOT(2,X+50,Y+50,X+50,Y);
140  PLOT(2,X+50,Y,X,Y);
150  PLOT(4,X+25,Y+25,X+25,Y+25);
160  WHILE GETKEY=CHR(0)DO;
170  HIRES(0);
190  END.
```

Figure 7.5 Filling an area with colour. This is not always straightforward, and you sometimes have to fill an area in several steps, using a different **X,Y** position each time.

switch off, just type KILL and press RETURN. This will switch the screen out of high-resolution mode into normal action, and will restore lower-case text if you have been using upper-case.

Figure 7.6 shows another example of high-resolution action, this time illustrating the 'unfill' action of a Mode 5 PLOT. In the main program, the instruction WINDOW(25) instructs the complete screen to be high resolution. By using a smaller number in the WINDOW instruction, it is possible to mix an ordinary text screen with a high-resolution screen. The number that follows WINDOW is the number of text lines (up to 25) which are occupied by the high-resolution screen. A WINDOW(12) takes up about half of the screen depth,

```
10  PROGRAM P7N7;
20  CONST RAD=0.017453;
30  VAR X,Y,R:INTEGER;
40  PROCEDURE CIRCLE(X,Y,R:INTEGER);
50  VAR J,OLDX,NEWX,OLDY,NEWY:INTEGER;
60    A:REAL;
70  BEGIN;
80  OLDX:=X-R;
90  OLDY:=Y;
100  FOR J:=1 TO 12 DO BEGIN
```

```
110   A:=J*RAD*30;
120   NEWX:=X-ROUND(R*COS(A));
130   NEWY:=Y+ROUND(R*SIN(A));
140   PLOT(2,OLDX,OLDY,NEWX,NEWY);
150   OLDX:=NEWX;
160   OLDY:=NEWY;
170   END;
180  PLOT(4,X,Y,X,Y);
190  END;
200  PROCEDURE GUN;
210  BEGIN;
220   PLOT(2,1,0,10,10);
230   PLOT(2,1,1,10,10);
240   PLOT(2,0,1,10,10);
250  END;
260  PROCEDURE SHOOT;
270  VAR J,X :INTEGER;
280  BEGIN;
290   J:=0;
300   FOR X:=10 TO 86 DO
310    BEGIN;
320    PLOT(2,X,X,X+1,X+1);
330    WHILE J<10 DO J:=J+1;
340    PLOT(3,X,X,X+1,X+1);
350   END;
360   PLOT(5,100,119,100,119);
370  END;
380  BEGIN;(*MAIN PROGRAM*)
390   HIRES(1);
400   WINDOW(25);
410   PAPER(7);
420   INK(2);
430   PLOT(0,0,0,0,0);
440   PLOT(1,0,0,0,0);
450   X:=100;
460   Y:=100;
470   R:=20;
480   CIRCLE(X,Y,R);
490   GUN;
500   SHOOT;
510   HIRES(0);
530  END.
```

Figure 7.6 Illustrating the clearing of an area.

and the high-resolution part is always the *upper* part. The important point about WINDOW is that though you will always get the effect of WINDOW(25) when the machine is first switched on, the effect of a smaller WINDOW space will persist until countermanded. If you have any programs that use windows, then it's advisable to use a WINDOW(25) or whatever you may need in other programs to avoid unwanted WINDOW effects in high-resolution graphics. We'll come back to the effect of WINDOW again later. Meantime, the program of Figure 7.6 has three procedures, CIRCLE, GUN and SHOOT, whose names describe roughly what they do. The CIRCLE routine draws a very rough approximation to a circle, using twelve straight sides, and fills this area. It's quicker, in fact, to go for a filled circle from the start, but this method will illustrate what is needed. When the circle is filled, the GUN procedure draws a miniature naval gun, and procedure SHOOT fires a projectile. When this touches the circle, at co-ordinates that have been calculated, the circle is wiped out, and the program ends. The wiping out action is done by the 'unfill' action in line 360.

In this example, it was possible to calculate from the geometrical arrangement when the 'shot' would just touch the circle. Suppose that this calculation is difficult or impossible. As it happens, there is a useful function EXAMINE which can be pressed into action in this case. Figure 7.7 shows how this works, using a modification of the previous example. The circle is drawn this time by plotting a set of lines, which is a quicker way of producing a filled circle. When the gun is fired, the position of the shell is tested continually by using EXAMINE. When a red object is reported just ahead of the shell, the 'wiping' operation is triggered. The syntax of EXAMINE can be seen in line 280. The SHOOT routine has been rearranged so that it depends for the result of EXAMINE to stop. This can be risky unless you know positively that there will be something to stop it, and in most cases it's a good idea to add some other stop conditions to the UNTIL line. You might, for example, use UNTIL (EXAMINE(X+2,X+2)=1) OR (X>255) in place of the simpler version that is shown. In this procedure, X is incremented, and the plot done, and then wiped until the space just ahead of the shell is found to be in INK colour. At this point, the value of EXAMINE, which has been 0 up to now, changes to being 1. This ends the loop, and the usual wipe action is performed and the program then ends.

GRAPH DRAWING

Because Pascal is a language that can have a lot of appeal for anyone who is programming for a wide range of purposes, the topic of graph drawing is one that is an obvious candidate for the high-resolution graphics of the C-64. In this respect, the ability to use a window of text is particularly helpful, because this allows you to display simultaneously the graph and an explanation of its meaning. Graph plotting is carried out using a mode 2 PLOT, and with the two

```
10  PROGRAM P7N7;
20  VAR X,Y,R:INTEGER;
30  PROCEDURE CIRCLE(X,Y,R:INTEGER);
40  VAR J,D:INTEGER;
50  BEGIN;
60  FOR J:=Y-R TO Y+R DO
70    BEGIN;
80    D:=ROUND(SQRT(SQR(R)-SQR(J-Y)));
90    PLOT(2,X-D,J,X+D,J);
100   END;
110 END;
120 PROCEDURE GUN;
130 BEGIN;
140   PLOT(2,1,0,10,10);
150   PLOT(2,1,1,10,10);
160   PLOT(2,0,1,10,10);
170 END;
180 PROCEDURE SHOOT;
190 VAR J:INTEGER;
200 BEGIN;
210 X:=10;
220 Y:=10;
230 REPEAT;
240   PLOT(2,X,X,X+1,X+1);
250   WHILE J<10 DO J:=J+1;
260   PLOT (3,X,X,X+1,X+1);
270   X:=X+1
280   UNTIL EXAMINE(X+2,X+2)=1;
290   PLOT(5,100,119,100,119);
300 END;
310 BEGIN;(*MAIN PROGRAM*)
320   HIRES(1);
330   WINDOW(25);
340   PAPER(7);
350   INK(2);
360   PLOT(0,0,0,0,0);
370   PLOT(1,0,0,0,0);
380   X:=100;
390   Y:=100;
400   R:=20;
410   CIRCLE(X,Y,R);
420   GUN;
430   SHOOT;
440   HIRES(0);
460 END.
```

Figure 7.7 Using **EXAMINE** to decide when a point is of **INK** colour.

sets of co-ordinates identical. The WINDOW allowance for graphing is obtained, as noted earlier, by using WINDOW followed by a number of text lines within brackets. Though the size of the window is given in text lines, the result is the number of lines that will be devoted to the high-resolution screen.

Figure 7.8 shows an example of this type of work. The main program sets the window size, paper and ink colours, and switches to high resolution. The procedures GRAPH and WORD then take care of the graph drawing and the text respectively. A PRESS ANY KEY step is used to hold the display while you inspect it. In this example, the graph has been drawn from a mathematical equation, but in practice it's more likely that you would either read in values from tape or disk or possibly supply them from the keyboard. This would call for a more elaborate graph-drawing procedure, because the range of values would have to be checked so that the graph filled its window reasonably well. Nothing looks more ridiculous than a graph which fills one corner of a screen, or one which displays only two points out of a total of twenty that were entered!

The details of the program of Figure 7.8 are useful to note if you are planning your own graph-drawing work. Notice, for example, that you can issue all of the WINDOW, PAPER, INK and the PLOT instructions that clear the high-resolution screen before you actually switch to higher resolution. Another point to note here, as in the previous example of Figure 7.6, is that a 'real' constant has to be written in the correct form. You can write .05236 as 0.05236 or as 5.235E-2, but there *must* be a digit before the decimal point. You get used to omitting this digit in BASIC, but Pascal is a lot more fussy. The graph routine is 'scaled', which means that the quantities that are being plotted have been multiplied so as to make the graph of a reasonable size. The sine of an angle can never exceed 1 (+ or −), nor can its cube, and a graph whose Y dimension changed only from +1 to −1 would not look particularly useful. By multiplying by 60, the scale in the Y direction is made to fill much more of the window. When you use windows, remember that the window size is counted in text lines, numbering from 1 to 25. Since 25 text lines fill a window whose maximum Y number is 200, then each text line is worth 8 steps in the Y direction. This is how you can calculate where to put the origin of the graph. In this example, using WINDOW(20), we have used 5 lines for text, which is a Y dimension of $5 \times 8 = 40$. The Y values should then start at 40 or more, and when we write text, we have to skip the first 20 lines. In fact, the graph origin is taken to be 2,42 so as to allow a little space all round. Lines 170 to 200 then draw a couple of graph axes, and also complete the square so as to show the area that will be used for graphing. If you don't do this, it always looks as if the graph is half-finished because of the limited range of the X direction. Note too the use of ROUND to round a number up or down to the nearest integer. We must use integers for the graph plotting instructions, and the ROUND instruction converts the REAL result of the sine cube calculation into integer form.

```
 10  PROGRAM P7N9;
 20  CONST MESG='PRESS ANY KEY TO END DI
SPLAY';
 30  PROCEDURE WORD;
 40  VAR  J:INTEGER;
 50   BEGIN;
 60    PAGE;
 70    FOR J:=1 TO 20 DO WRITELN;
 80    WRITELN('GRAPH DEMONSTRATION':29)
;
 90    WRITELN('GRAPH OF SINE CUBED WAVE
':32);
 100  END;
 110  PROCEDURE GRAPH;
 120  CONST YO=120;
 130        RO=0.05236;
 140  VAR X,Y:INTEGER;
 150        D:REAL;
 160   BEGIN;
 170    PLOT(2,2,42,255,42);
 180    PLOT(2,2,42,2,200);
 190    PLOT(2,2,199,255,199);
 200    PLOT(2,254,42,254,200);
 210    FOR X:=1 TO 240 DO
 220     BEGIN;
 230      D:=SIN(RO*X);
 240      Y:=YO+ROUND(60*D*D*D);
 250      PLOT(2,X,Y,X,Y);
 260     END;
 270   END;
 280  BEGIN;(*MAIN PROGRAM*)
 290   WINDOW(20);
 300   PAPER(4);
 310   INK(7);
 320   PLOT(0,0,0,0,0);
 330   PLOT(1,0,0,0,0);
 340   HIRES(1);
 350  WORD;
 360  GRAPH;
 370   WRITELN;
 380   WRITELN(MESG:34);
 390   WHILE GETKEY=CHR(0) DO;
 400   HIRES(0);
 420  END.
```

Figure 7.8 Drawing a graph on the high-resolution screen. The slow action is due to the calculations. If you had entered the graph data as number arrays, then the graphing action would be very rapid.

LOWER RESOLUTION

In all this talk of high-resolution graphics, it's as well to remember that many display actions do not need high resolution. A good example is the use of bar-graphs. A bar-graph can be more easily displayed in ordinary text graphics than in high resolution, and the results are just as good, perhaps better, and certainly quicker. Using text graphics, the ordinary graphics mode of the C-64, allows you to control the display much more easily, and to mix text and graphics more easily. In this you are aided by an extension to standard Pascal in the form of the VDU instruction. This is an instruction which is peculiar to Oxford Pascal, and which performs in a similar way to the VDU instruction of the BBC Micro. In Oxford Pascal, VDU is used with the syntax VDU(Y,X,C), in which X and Y are co-ordinates for screen position (integers) and C is a character which can be written, for example, in ASCII code such as CHR(63) or in the form '?', using a single inverted comma on each side of the character that you want to use. Note that the Y number comes ahead of the X number. The manual shows this as VDU(X,Y,C) but then states that X is the row number and Y the column number. It's easier to remember how it works if you keep to the conventional X and Y meanings.

Figure 7.9 shows how a set of bar-graphs can be drawn with impressive speed from a set of input numbers. The situation is an imaginary report on milk yield for a complete year, and the numbers which are entered are in the range 1 to 22. In a real-life situation, real numbers would be entered, multiplied by a constant and rounded so as to give a final range of 1 to 22. This range has been picked so that the graphs fall conveniently above the text and can fill the screen at the maximum value. As usual, we start looking at the main program first. This assigns values for wording to be written down and along the graph axes, and then calls the procedure MILKDATA. This procedure obtains the items of data, and converts them into the form that will be suitable for the bar-graph. SCREEN, PEN and BORDER colours are then chosen, the screen is cleared, and the graphs, along with their labelling, are drawn.

Lines 410 to 450 put in the vertical lettering which shows what is being measured on the vertical axis. This makes use of the VDU instruction which can place characters anywhere on the screen and in any order. The characters are read out of the packed array VERTMES by using Y−6 as the array item number. Since the range of Y is from 7 to 16, using Y−6 gives 1 to 15, the correct range of letter position numbers in VERTMES. With this vertical lettering in place, the next step is to put in the abbreviated names for the months along the X axis. This is done in very much the same way in lines 480 to 510, and then the label word MONTHS is placed under the range of months by lines 540 to 570. The screen is then ready for drawing the vertical bars, using the character whose ASCII code is 166.

This is done in lines 580 to 640. There are twelve bars, so a loop which counts from 1 to 12 is used, and for each month, a number of the characters which form

```
 10  PROGRAM P7N10;
 20  CONST MESG1='PLEASE INPUT DATA AS R
EQUESTED';
 30        MESG2='1 TO 22 ONLY';
 40  VAR X,Y:INTEGER;
 50     MILDAT:ARRAY[1..12]OF INTEGER;
 60    VERTMES:PACKED ARRAY[1..10] OF CH
AR;
 70     HORMES:PACKED ARRAY[1..6]OF CHAR
;
 80    HORSCAL:PACKED ARRAY[1..36]OF CHA
R;
 90  C:CHAR;
 100  PROCEDURE MILKDATA;
 110  VAR J,N,K:INTEGER;
 120   BEGIN;
 130   PAGE;
 140   WRITELN('MILK DATA':35);
 150   WRITELN;
 160   WRITELN(MESG1:35);
 170   WRITELN(MESG2:26);
 180   FOR J:=0 TO 11 DO
 190    BEGIN;
 200  REPEAT;
 210    FOR N:=1 TO 3 DO
 220     BEGIN;
 230      WRITE(HORSCAL[3*J+N]);
 240     END;
 250     WRITE(' ');
 260      READLN(K);
 270     UNTIL (K>0) AND (K<23);
 280     MILDAT[J+1]:=22-K;
 290    END;
 300   END;
 310  BEGIN;(*MAIN PROGRAM*);
 320  VERTMES:='MILK YIELD';
 330  HORMES:='MONTHS';
 340  HORSCAL:='JANFEBMARAPRMAYJUNJULAUG
SEPOCTNOVDEC';
 350   MILKDATA;
 360  SCREEN(4);
 370  PEN(7);
 380  BORDER(2);
 390  PAGE;
```

```
400   X:=1;
410   FOR Y:=7 TO 16 DO
420    BEGIN;
430    C:=VERTMES[Y-6];
440    VDU(Y,X,C);
450    END;
460   Y:=23;
470   FOR X:=2 TO 37 DO
480    BEGIN;
490    C:=HORSCAL[X-1];
500    VDU(Y,X,C);
510    END;
520   Y:=24;
530   FOR X:=17 TO 22 DO
540    BEGIN;
550    C:=HORMES[X-16];
560    VDU(Y,X,C);
570    END;
580   FOR X:=1 TO 12 DO
590    BEGIN;
600    FOR Y:=MILDAT[X] TO 22 DO
610     BEGIN;
620     VDU(Y,3*X,CHR(166));
630     END
640    END;
650   WHILE GETKEY=CHR(0) DO;
670   END.
```

Figure 7.9 Drawing bar-graphs on the low-resolution screen. This program illustrates the very useful **VDU** instruction, which is not standard Pascal.

the bar must be printed vertically. The number that has to be used in the loop that prints the vertical characters is obtained from the array MILDAT, using the correct array item, MILDAY[X]. The loop that starts in line 600 prints the correct number of blocks, and the horizontal position is given by using 3*X in the VDU instruction. Note that we have used CHR(166) as the character in the VDU instruction.

All we need to inspect now is the MILKDATA procedure which gets the numbers for the bar printing loop. The important part of this procedure lies in lines 180 to 290. The main loop starts with J:=0 to 11, giving a total of twelve entries. Within this loop, another loop prints the abbreviation for a month and prompts for a reply until an integer in the correct range is entered. The month abbreviation is printed by using VDU to write three characters taken from array HORSCAL. The formula 3*J+N ensures that the first set will be characters 1, 2

and 3, the next set 4, 5, and 6, and so on. The number that is input must be between 1 and 22. This is because line 280 transforms the number into the form that is needed for drawing the bars by subtracting it from 22. If you enter 21, for example, the end result in array MILDAT will be 1, and this allows a bar to be drawn from Y=1 to Y=22 when the drawing part of the program operates. For a real-life case, the entry would be more carefully checked, the range would have to be known in advance, and each number then multiplied by a scale factor. It is possible to enter any numbers into a temporary array, find the minimum and maximum values, and then use the program to calculate a scale factor and apply it. For most purposes, however, a program will be written to solve a particular problem, and the range of likely values will be known.

SOUND MANAGEMENT

The use of sound on the Commodore 64 is as frustrating to the BASIC programmer as the use of high-resolution graphics. Despite the fact that the Commodore 64 has one of the best sound-producing systems available on any small computer, few users ever achieve mastery of it. This is because sound, like high-resolution graphics, is obtained by using POKE instructions in BASIC, or by the use of machine code. This makes any experimenting with the sound instructions very tedious, because unless you become very familiar with the memory addresses that have to be poked, and with the range of numbers that must be used, you are unlikely to be quite sure what POKE instruction achieved what effect. Oxford Pascal allows extensions of standard Pascal instructions which enable you to take rather more effective command of the sound system of your C-64, though not to the exent that some specialised 'sound-editor' programs permit.

Before we can look at what instructions are available, we need to know something about sound itself. The ability to produce sound is an essential feature of all modern computers. The sound of the Commodore 64 computer comes from the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers. In addition, the Commodore 64 allows you to create sound effects, depending on whether you want just a reminder, or a melody, or a pistol shot.

What we call sound is the result of rapid changes of the pressure of the air round our ears. Everything that generates a sound does so by altering the air pressure, and Figure 7.10 shows how the skin of a drum does this. All other musical instruments also rely on the principle of something which vibrates, and pushes the air around. Air pressure, however, is invisible, and we don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or hertz. A cycle of any wave is a set of changes, first in one direction, then in the other and back to normal, which we can illustrate

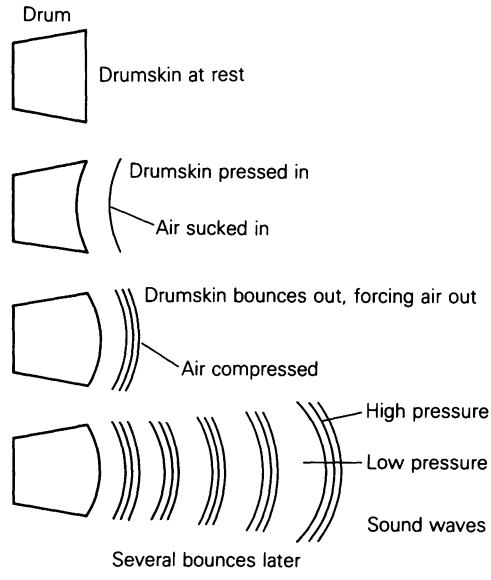

Figure 7.10 How the skin of a drum produces sound waves.

by the graph in Figure 7.11. The reason that we talk about a sound 'wave' is because the shape of this graph is a wave shape.

The frequency of sound is its number of hertz—the number of cycles of changing air pressure per second. If this amount is less than about 20 hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15 000 hertz. The frequency of the waves corresponds to what we sense as the 'pitch' of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high-pitch treble note. Human ears are not sensitive to sounds whose frequency is above 20 000 hertz (called 20 kilohertz), but many animals can hear sounds in this range.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of 'amplitude', which is the maximum change of pressure of the air from its normal value. The shape of the wave, which is the shape of the graph of amplitude plotted against time, decides on the 'quality' of the sound, whether it seems harsh or smooth. For complete control over the generation of sound, then, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds. We'll start with the elementary problems of producing notes which make up a musical melody. The main task here is to translate from written music into the instructions of a Pascal program.

If you have no experience of music, however, this may seem rather puzzling

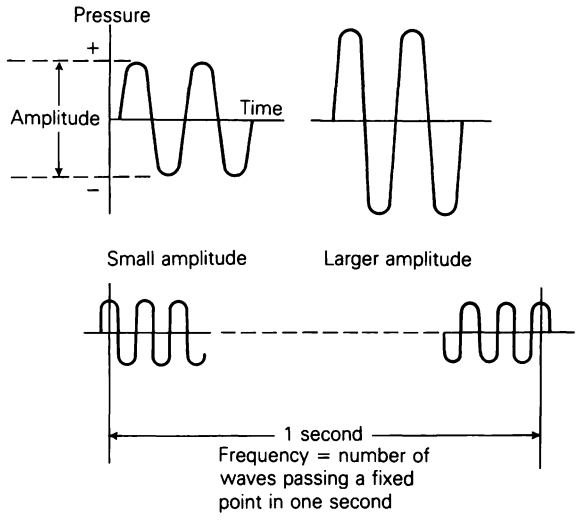

Figure 7.11 A graph of air pressure against time, showing how the pressure changes in a wave pattern.

to you. How do we go about writing down music? For each note, we have to specify what the note is (its pitch), how loud it must be, and for how long it is to be played. In written music, this is done by using a type of chart for the pitch, and different shapes of markings (notes) for the duration. Loudness is indicated by using letters such as *f* (loud) and *p* (soft). More than one letter can be used, so that *fff* means very loud, and *ppp* means very soft. Each sound is indicated by a note, a shape on the chart, and the shape of the note gives some information about the duration of the note. In addition to this, each piece of music will start with some advice about the speed at which the notes are to be played. One of these methods is a metronome reading. The metronome is a gadget which ticks at regular intervals, and the metronome reading for a piece of music is the number of metronome ticks per minute. A more ancient way of indicating speed is the use of Italian words like 'allegro' (fast), 'lento' (slow) and so on. What these speed settings decide is how many unit notes will be played in a minute. The unit note is the crotchet, so if a piece of music is marked at a metronome speed of 60 (pretty slow), then there will be 60 crotchets played per minute. The durations of all the other notes are decided in comparison to this unit, the crotchet. A minim sounds for twice as long as the crotchet, a semibreve for twice as long as a minim, which is four times as long as the time of a crotchet. The quaver sounds for only half the time of the crotchet. A semiquaver sounds for only half the time of a quaver, which is quarter of the time of a crotchet. The crotchets and other timed notes are indicated by the shapes of the written notes, as Figure 7.12 shows. In addition, symbols are used to indicate

Symbol	Time	Name
𝅘𝅥𝅯	1/8	Demisemiquaver
𝅘𝅥𝅰	1/4	Semiquaver
𝅘𝅥𝅮	1/2	Quaver
𝅘𝅥	1	Crotchet
𝅗𝅥	2	Minim
𝅝	4	Semibreve

Figure 7.12 The timed notes of written music.

silences in the music, and these are based on the same idea of a unit duration of silence, and others which are twice, four times, half, or quarter. These silence marks are shown in Figure 7.13.

The pitch of a note is indicated in written music by placing it on a kind of musical map which is called the 'stave' (Figure 7.14). Piano music uses two of these staves, each consisting of five lines and four spaces. The upper stave is the treble stave, and it is used for writing the higher notes which will be played on the piano with your right-hand. The lower stave is the bass stave, the lower notes, played with the left-hand. Instruments which do not use a keyboard will normally have music written with only one stave. In addition to this representation of notes by position on staves, we also use the letters of the alphabet from A to G to name the notes.

Rest symbol	Time
𝄾	1/4
𝄿	1/2
𝄽	1
𝄼	2
𝄻	4

Figure 7.13 The silence marks in written music.

The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C major'. The scale starts on a note that is called 'middle C', and ends on a note that is also called C, but which is the eighth note (counting inclusively) above middle C. A group of eight notes like this is called an 'octave', so that the note you end with in this scale is the C which is one octave above middle

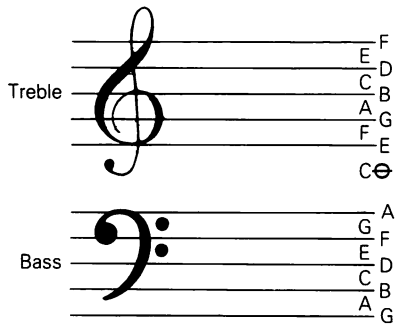

Figure 7.14 The stave of written music, which is a type of map for the notes.

C. Because music (in the Western hemisphere, at least) is based on this group of eight notes, we use only the first seven letters of the alphabet in naming the notes. Why seven? Well, the eighth note is the end of one octave and the start of the next, so it bears the same name. The scientific basis of all this is that if you take middle C, and find the frequency of the sound of this note, then the C which is the next octave above middle C has precisely double the frequency value of middle C. The C below middle C has half the frequency of middle C, and so on. That's why the ancient Greeks always thought that music was a branch of mathematics.

The appearance of these keys on the piano keyboard is illustrated in Figure 7.15. Middle C is, logically enough, at the centre of the keyboard and we move right for higher notes, left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a 'tone' and the smaller spacing is called a 'semitone'. Each scale will contain two semitones. On written music, middle C appears midway between the treble and bass staves.

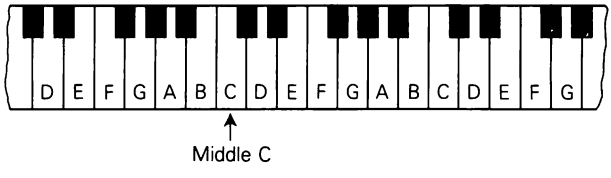

Figure 7.15 The position of named notes on the piano keyboard.

It's time now to look at some more revelations. What's the difference between a piano and a clarinet? Yes, I know, you'd look silly blowing a piano, but what's the difference in the sound? You can tell which of these instruments is playing a note, even if it's the same note at the same volume. One answer is

that the shape of the waves is different. Some method of controlling the shape of the waves, then, is an essential part of any music synthesiser. The odd thing is that very few computers have any simple way of doing this. The Commodore 64 allows you to select from a few wave shapes, enough to allow it to be able to imitate a very wide range of instruments. The other important control that you have over the waves is to alter the 'envelope' of a note.

The word 'envelope' needs some explanations. Envelope means the pattern of a note. A musical note does not consist of just one sound wave, but of many. While a note is sounding, this volume need not be constant, though it is for the traditional 'electric-organ' type of note. For example, when you strike a piano key, the note starts very loud, and its volume then dies away as the string vibration dies away. Its envelope is therefore like the shape that is shown in Figure 7.16—rising very sharply, then fading away. Each musical instrument produces its own type of envelope, and for some instruments, the effort of the player can alter the shape of the envelope. It's never easy to design your own shapes of envelopes with a computer, but the Commodore 64 uses a useful scheme which is followed by many synthesisers.

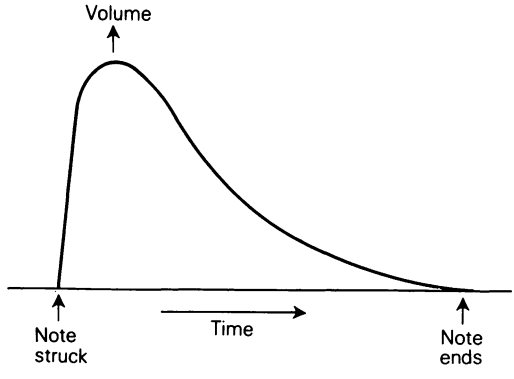

Figure 7.16 The 'envelope' of a piano note.

Take a look at Figure 7.17. This shows an envelope which is typical of many musical notes. It has a sharply rising amplitude at the start, which we call the 'attack'. This is followed by the 'decay' portion, in which the amplitude drops. The amplitude then remains steady for a time, in the 'sustain' section, and then finally drops to zero in the 'release' section. Oxford Pascal for the Commodore 64 gives us the chance to synthesise even a waveform like this by using an ENVEL instruction. The ENVEL instruction has to be followed by five integers, expressing voice or channel number (1 to 3), attack rate (0–15), decay rate (0–15), sustain time (0–15) and release rate (0–15).

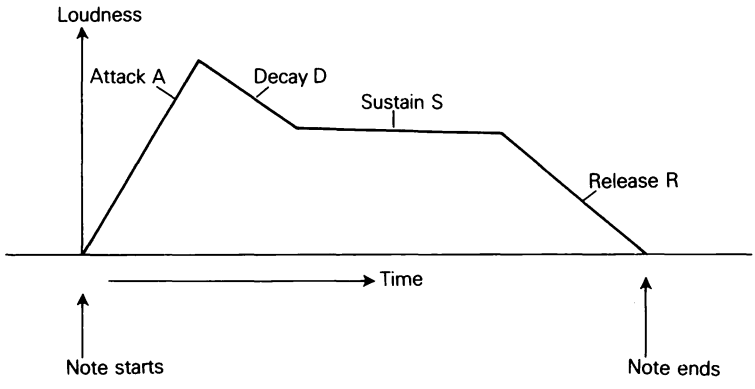

Figure 7.17 The four sections of an envelope.

OXFORD PASCAL SOUND

There are two main instructions, ENVEL and VOICE, for producing sound, and one control, VOLUME, which does as it says. You can, of course, control the *absolute* volume of a note by using the volume control of the TV receiver, but VOLUME allows you to program the volume of a note while it is playing. One complication that you have to watch out for is that the computer stores the results of any instructions. You can therefore find that the instructions which produced a sound are still in place unless you have switched off, and these stored instructions will affect the next sound.

It's time for some illustrations. Figure 7.18 shows a sound program which simply plays an ascending sequence of notes. Now this is not a musical scale (musicians might not think it is a musical anything!), because obtaining a musical scale is not quite so easy. For a sequence of notes like this, the easiest way of

```
10  PROGRAM P7N19;
20  VAR J,F:INTEGER;
30  BEGIN;
40  VOLUME(15);
50  ENVEL(1,0,0,15,15);
60  FOR J:=0 TO 7 DO BEGIN;
70   F:=4295+536*J;
80   VOICE(1,F,1,10000);
90   END;
110  END.
```

Figure 7.18 A sequence of notes which are controlled by the **F** variable in the **VOICE** statement.

getting frequency numbers is by use of a formula, as shown in line 70. This, however, can't produce a musical scale, because the musical scales that sound correct to our ears are much more complicated, and not so easy to obtain simply by a formula. The numbers that you have to use for musical notes are obtained from the numbers in the Commodore 64 BASIC manual. The manual shows each note as using a 'Hi Freq' and a 'Low Freq' number. If you multiply the high frequency number by 256 and add to this the low frequency number, you will get the frequency number to use in the Oxford Pascal VOICE instruction. For example, middle C is shown in the Commodore 64 manual as C4 with a high number of 17 and a low number of 37. The number to use in the VOICE instruction is therefore 245*17 + 37, which is 4389. Because Pascal does not have the useful READ...DATA instruction of BASIC, it's not quite so easy to type out a list of numbers and use them. You can, however, easily write a program that will accept a set of integers and store them in an array to use as frequency numbers. In this way, it's easier to write in your own notes, and if the program allows storage of the array on tape or disk, you have the makings of a music program.

EXPLORING SOUNDS

The ease with which you can experiment with the sound of the Commodore 64 using Oxford Pascal allows you to do easily things that are rather hard to achieve in BASIC. One useful thing is to check what the effect of different values might be. Figure 7.19 shows such a program which illustrates the rather subtle effects of different A and D numbers. In a program like this, you sometimes find that certain combinations of numbers cause the sound to shut off,

```
10  PROGRAM P7N20;
20  VAR A,D:INTEGER;
30  BEGIN;
40   REPEAT;
50  POKE(54276,254);
60   VOLUME(15);
70   WRITE('ENTER A AND D NUMBERS');
80   READLN(A,D);
90   ENVEL(1,A,D,1,1);
100   VOICE(1,4000,0,9000);
110   UNTIL FALSE;
130  END.
```

Figure 7.19 A program that lets you hear the effects of different **A** and **D** numbers. If you do not include line 50, you may find that you get silence, or no effect of varying the **A** and **D** numbers.

and you can also find that the program works once perfectly and thereafter not again. This is because a vital address has not been altered by the Pascal instructions, and in the program of Figure 7.19, this is done in line 50. The POKE instruction is in its Oxford Pascal form, which uses brackets surrounding the numbers, and the address 54276 is for voice 1. For the other two voices, the addresses are 54283 and 54290 respectively. When you make full use of A, D, S, and R numbers, you have to match up the duration of the note as well, because a very long duration will mask the effects of the envelope.

In general, the use of sound effects depends on a lot of practice, and a lot of listening! There are many published examples in magazines and in books of sound programs for the C-64, and adapting these for Oxford Pascal is not particularly difficult—the tables below show how the quantities can be translated. You will find, however, that if you want to make use of some of the advanced features of the sound system, such as filtering, you will have to resort to POKE instructions again, and this makes the sound instructions of Oxford Pascal rather less satisfactory than the high-resolution graphics instructions.

BASIC	**O.P.**
POKE 54272,N	Frequency number in
POKE 54273,N	VOICE instruction.
POKE 54276,N	Wave-type number in VOICE.

(Note: The number in this register must be odd if the ADSR system is to operate.)

POKE 54277,N	A and D numbers of ENVEL. The number that is poked is 16*A+D.
POKE 54278,N	S and R numbers of ENVEL. The number that is poked is 16*S+R.

For second channel, use POKE address numbers + 7. For third channel, use POKE address numbers + 14.

POKE 54296	VOLUME number.

Several other address numbers are preset in Pascal, but can still be reached by POKE instructions. Remember that the Pascal version of POKE requires address and data in brackets.

8

STRING PROCEDURES AND FUNCTIONS

Compared with BASIC, standard Pascal seems rather weak on string-handling routines. This is particularly true when Pascal is compared to some of the later versions of Microsoft BASIC, as used, for example, in the MSX micros and also in the IBM PC. Some varieties of Pascal, notably the Pascal for the IBM PC, have extended the language considerably to allow the types STRING and LSTRING, and to include a number of procedures which can manipulate these strings. Oxford Pascal sticks closer to the standard in this respect, so that each string has to be represented by a packed array of CHAR. When you have moved from BASIC to Pascal, however, this can leave you wondering how to implement actions, such as TAB, which you always took for granted with BASIC. You also have to learn how to make procedures for things like LEFT$, MID$ and RIGHT$, but you will find that a few actions, notably the VAL action, are very much more difficult to implement because Pascal is much more fussy about variable types than BASIC. This chapter will be devoted to procedures which help you to bridge that gap. A named Pascal procedure has the great advantage over a BASIC subroutine in that it is called into action by using its name rather than by the use of a line number. Because of this, Pascal procedures can be used as if they were new words in the language, which, for you, they are. The one action that is missed most by ex-BASIC users, however, is assigning a string variable to some text, as A$="THIS IS A STRING". This is very difficult to implement in Oxford Pascal because of the requirement that the length of a string must always match the length of the array that has been declared for it.

The set of string actions starts with Figure 8.1. I have split these into sections rather than showing one long listing, because it's easier this way to select the actions that you need most frequently. The simplest string action consists of preset tabbing. By defining constant TAB of eight spaces, we can place the word TAB into a WRITE statement and automatically space words or numbers.

```
 10  PROGRAM P8N1;
 20  CONST MAX=80;
 30  TAB='        ';(*8 SPACES*)
 40  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 50  VAR TEXT,SPACE:STRING;
 60  FUNCTION LEN(S:STRING):INTEGER;
 70  VAR J:INTEGER;
 80   BEGIN;
 90   J:=MAX;
 100   WHILE S[J]=' 'DO J:=J-1;
 110  LEN:=J;
 120  END;
 130  PROCEDURE SPC(N:INTEGER);
 140  VAR J:INTEGER;
 150  BEGIN;
 160   FOR J:=1 TO MAX DO BEGIN
 170   SPACE[J]:=' ';END;
 180   WRITE (SPACE:N);
 190  END;
 200  BEGIN;(*MAIN PROGRAM*)
 210  PAGE;
 220  WRITELN;
 230  WRITELN('A',TAB,'B',TAB,'C',TAB,'D
');
 240  WRITELN;
 250  WRITELN('YOUR STRING, PLEASE');
 260  READLN(TEXT);
 270  WRITELN(TEXT,' CONSISTS OF ',LEN(T
EXT):2,' CHARACTERS');
 280  WRITELN;
 290  WRITE('THIS');SPC(5);
 300  WRITE('IS');SPC(3);
 310  WRITE('SPC');SPC(4);
 320  WRITE('IN');SPC(7);
 330  WRITE('ACTION');
 350  END.
```

Figure 8.1 The function and procedure for string actions **LEN** and **SPC**, and the use of a preset **TAB** constant.

Remember, however, that this does not override the field requirements, and that typing WRITELN('TOTAL',TAB,A) where A is a real number will produce more than eight spaces if A is not a large number. If you field your numbers correctly, however, this will not be a problem. The use of this preset TAB, which can, of course, be of five spaces or any other number you please, gets over a lot of the display problems that ex-BASIC programmers encounter when they start to use Pascal. In this sense, TAB is being used more like the TAB of a typewriter than the TAB of BASIC. The action is closer to the instructions SPC(8) which is allowed in some versions of BASIC. Remember that Pascal can use the field numbers for its tabulation, and these are often easier to use than the BASIC TAB instruction.

The next action in the list is the LEN function. This is a defined function which produces as a value for LEN the number of useful characters in a string. This is not the same as the total length of the string, because many strings will be padded out with blanks. This works in exactly the same way as the SHRINK procedure which we used earlier. The integer variable J is set to the maximum length of a string, which is defined throughout these functions and procedures as 80 characters. J is then decremented for as long as only spaces are found at the end of the string, so that the final value of J gives the length of the string. This number can be used in fielding, so that the spaces at the end of a string are not printed, as we used SHRINK. In addition, other procedures and functions can make use of this one, so that this function is one of our 'bankers', one that you should not delete unless you are sure that it is not used elsewhere.

The procedure SPC is one that will write a specified number of spaces. Note that unlike a function, a procedure can carry out the action of writing a string of spaces. A function cannot be used because a function in Pascal cannot return a string, only integers, reals and pointers. It's just possible to implement the action using a pointer, but hardly worth the extra programming. In this case, the variable SPACE is filled with spaces, and then written with the field size N. This integer is the value which has been passed to the procedure, so that the action is one of writing N spaces. It would be more useful if SPC could be used inside a WRITE set of brackets, but this is next best, and the syntax is illustrated in the main program.

The main program illustrates the use of this 'starter-pack' of string-handling procedures and functions. In line 230, the letters A, B, C, and D are tabbed across the screen, using the constant TAB. A figure of eight spaces is a particularly convenient one for most purposes, but you might want to use five for separating integers, particularly if the field size was small. Remember that this TAB, unlike TAB in BASIC, is really a spacing command. In other words, it sets the amount of space between items rather than the absolute position of items on the line. Line 270 illustrates the LEN function being used simply to print out the length of a string. This would not normally be of interest but you might want to make use of LEN to decide whether a string was too long to use, or in other string-handling actions. The result of LEN can always be fielded to 2

when MAX=80. The system of making strings equivalent to packed arrays of characters, however, means that you can have strings as long as you like, limited only by the memory of the computer. This is not something that most users will need, but it's useful to know, because it can be convenient to hold string data in the form of one long string. Finally, lines 290 to 320 illustrate the use of SPC. This, remember, is a procedure, not a function, and it has to occupy a space of its own in a program. Unlike TAB in BASIC, for example, which can be part of a PRINT statement, SPC can't be made part of a WRITE statement. You can, however, place it on the same line as a WRITE, separated by a semicolon. This avoids the clumsy appearance of having separate lines for SPC. You can, if you like, write a whole Pascal program in one line provided that you put in the semicolons at the correct places. This is something that I have not illustrated and would not want to encourage, because it makes a program very difficult to read.

A procedure which resembles the PRINTAT facility of some computers is illustrated in Figure 8.2. In the BASIC of many computers, PRINTAT is followed by X and Y co-ordinates, and then by a string which is to be printed. This allows a string to be printed anywhere on the screen, irrespective of the normal position of the cursor. PRINTAT can be implemented in Oxford Pascal thanks to the provision of VDU, and the PRINTAT action is almost as fast as normal printing. The procedure makes use of LEN to find the length of the text, so the LEN function must also be present if you want to use PRINTAT. The procedure requires three parameters, the X and Y positions (integers) and the string text. J is used to count characters from the string, and L carries the length number. The first character is printed at the correct position by using X and Y in the VDU statement, and the character is selected by using J as the array number. X and J are then both incremented, and J is compared to L. It's important to use L here rather than LEN(TEXT), because if the LEN function has to be called each time the loop is executed it will make the action very slow. Never put any function or procedure into a loop unless it is absolutely necessary, because the speed disadvantage can be enormous. If you want to see this, try the example, and then put LEN(TEXT) into the loop and note the change in speed.

The use of PRINTAT gives you a lot more freedom over the placing of text on the screen, and saves having to make each positioning of text into a major exercise. Useful additions, when you don't necessarily want the complete freedom of PRINTAT are VSPC and CENT. Procedure VSPC gives, as the name suggests, a number of empty lines, so spacing your text vertically. The CENT procedure will centre a title in its line. As always, you have to be careful of string dimensions. For the CENT procedure, it's better to work with character arrays of no more than 40 in length, because you wouldn't want to have two-line titles centred. Note therefore that the LEN procedure has been changed in this example, because if LEN is arranged for strings of 80 characters, it won't work on strings of 40 characters. It sounds crazy, and that's why some versions of Pascal have a type LSTRING which has a maximum length, but which has no

```
 10  PROGRAM P8N2;
 20  CONST MAX=80;
 30  TAB='        ';(*8 SPACES*)
 40  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 50  VAR TEXT,SPACE:STRING;
 60  X,Y:INTEGER;
 70  FUNCTION LEN(S:STRING):INTEGER;
 80  VAR J:INTEGER;
 90   BEGIN;
 100   J:=MAX;
 110   WHILE S[J]=' 'DO J:=J-1;
 120  LEN:=J;
 130  END;
 140  PROCEDURE SPC(N:INTEGER);
 150  VAR J:INTEGER;
 160  BEGIN;
 170   FOR J:=1 TO MAX DO BEGIN
 180   SPACE[J]:=' ';END;
 190   WRITE (SPACE:N);
 200  END;
 210  PROCEDURE PRINTAT(X,Y:INTEGER;TEXT
:STRING);
 220  VAR J,L:INTEGER;
 230  BEGIN J:=1;
 240  L:=LEN(TEXT);
 250  REPEAT
 260   VDU(Y,X,TEXT[J]);
 270   J:=J+1;X:=X+1;
 280   UNTIL J>L;
 290  END;
 300  BEGIN;(*MAIN PROGRAM*)
 310  PAGE;
 320  WRITELN('PLEASE INPUT YOUR TEXT');
 330  READLN(TEXT);
 340  WRITELN('..AND THE X,Y POSITION');
 350  READ(X,Y);
 360  PRINTAT(X,Y,TEXT);
 380  END.
```

Figure 8.2 Adding a procedure for **PRINTAT**, which allows you to print anywhere on the screen.

minimum, and which can be assigned with strings whose length is less than the maximum.

In any case, the full listing is shown in Figure 8.3. The VSPC procedure takes as its parameter the number of lines that you want to space vertically, and a

```
 10  PROGRAM P8N3;
 20  CONST MAX=80;
 30  TAB='        ';(*8 SPACES*)
 40  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 50     LINE=PACKED ARRAY[1..40]OF CHAR;
 60  VAR TEXT,SPACE:STRING;
 70     TITLE:LINE;
 80  X,Y:INTEGER;
 90  FUNCTION LEN(S:LINE):INTEGER;
 100  VAR J:INTEGER;
 110   BEGIN;
 120   J:=40;
 130   WHILE S[J]=' 'DO J:=J-1;
 140  LEN:=J;
 150  END;
 160  PROCEDURE SPC(N:INTEGER);
 170  VAR J:INTEGER;
 180  BEGIN;
 190   FOR J:=1 TO MAX DO BEGIN
 200   SPACE[J]:=' ';END;
 210   WRITE (SPACE:N);
 220  END;
 230  PROCEDURE VSPC(N:INTEGER);
 240  VAR J:INTEGER;
 250  BEGIN;
 260   FOR J:=1 TO N DO
 270   WRITELN;
 280  END;
 290  PROCEDURE CENT(TITLE:LINE);
 300  VAR J,L:INTEGER;
 310  BEGIN;
 320  L:=LEN(TITLE);
 330  J:=100-ROUND(L/2);
 340  WRITELN(TITLE:J);
 350  END;
 360  BEGIN;(*MAIN PROGRAM*)
 370  PAGE;
```

```
 380  WRITELN('PLEASE INPUT YOUR TITLE')
;
 390  READLN(TITLE);
 400  WRITELN('..AND THE NEXT VERTICAL P
OSITION');
 410  READ(Y);
 420  PAGE;
 430  CENT(TITLE);
 440  VSPC(Y);
 450  WRITELN('THIS IS WHERE THE NEXT TE
XT WILL BE');
 470  END.
```

Figure 8.3 The procedures for **VSPC** and **CENT**.

loop counts out an equal number of WRITELN statements. The CENT procedure takes a title, of type LINE. This has been defined as a packed array of 1..40 characters, and FUNCTION LEN has been altered to suit. The length of the useful part of the string is measured using LEN, and the field number for the centre position is calculated and assigned to J. The calculation is not the usual 20+(length)/2, because the normal field length of the string is 80, and if we print an 80-character string with a field of, say, 26, then all we do is to chop off 26 spaces from the end of the string. What we have to do is to add the field size found for 20+(length)/2 to the 80 character field that exists already, making the formula into 100+(length)/2. The ROUND action avoids problems with fractions—TRUNC would be just as appropriate as ROUND in this case. Line 340 could be amended to WRITE in place of WRITELN if you like, because it will not be possible to write immediately following the title in any case because of the spaces in the array.

LEFT, RIGHT AND MIDDLE

The BASIC functions LEFT$ and RIGHT$ are replaced to some extent in Oxford Pascal by the use of field numbers when a part string is printed out. For example, WRITE(TEXT:2) will print the first two characters on the left of a string, and WRITE(TEXT:−2) will print the last two on the right-hand side. These actions apply to printing only, however, and they don't allow you to do what you can do so easily in BASIC, that is to assign part of a string to another string variable name. The next few procedures, then, are aimed at providing this action, and also with providing the equivalent of a MID$ action.

Figure 8.4 tackles the LEFT and RIGHT problem. Now these are 'skeleton' procedures in the sense that they do just the minimum that is required. There is

```
 10  PROGRAM P8N4;
 20  CONST MAX=80;
 30          LL=40;
 40  TAB='        ';(*8 SPACES*)
 50  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 60      LINE=PACKED ARRAY[1..40]OF CHAR;
 70  VAR TEXT,SPACE:STRING;
 80      NEWTX:LINE;
 90  X,Y:INTEGER;
 100  FUNCTION LEN(S:STRING):INTEGER;
 110  VAR J:INTEGER;
 120   BEGIN;
 130   J:=MAX;
 140   WHILE S[J]=' 'DO J:=J-1;
 150  LEN:=J;
 160  END;
 170  PROCEDURE LEFT(TEXT:STRING;S:INTEG
ER);
 180  VAR J:INTEGER;
 190  BEGIN;
 200   J:=0;
 210   REPEAT J:=J+1;
 220   NEWTX[J]:=TEXT[J];
 230   UNTIL J=S;
 240   REPEAT J:=J+1;
 250   NEWTX[J]:=' '
 260   UNTIL J=LL;
 270  END;
 280  PROCEDURE RIGHT(TEXT:STRING;S:INTE
GER);
 290  VAR J,L,N:INTEGER;
 300  BEGIN;
 310   J:=0;L:=LEN(TEXT);
 320   REPEAT J:=J+1;
 330   NEWTX[J]:=TEXT[L-S+J];
 340   UNTIL J=LL;
 350  END;
 360  BEGIN;(*MAIN PROGRAM*)
 370  PAGE;
 380  WRITELN('PLEASE INPUT YOUR TEXT');
 390  READLN(TEXT);
 400  WRITE('THE LEFT(3) SLICE IS ');
 410  LEFT(TEXT,3);WRITELN(NEWTX);
```

```
420  WRITELN;
430  WRITE('AND THE RIGHT(3) IS ');
440  RIGHT(TEXT,3);
450  WRITELN(NEWTX);
470  END.
```

Figure 8.4 **LEFT** and **RIGHT** string extraction procedures.

no checking to see if the numbers that you use are sensible, and standard string lengths of 80 and 40 have been assumed. What is being done is to read the chopped piece of string into a string which is defined as a shorter array. In the examples, a string which is an 80-character array is chopped and the chopped portion assigned into a string which is a 40-character array. Two rather different techniques for adding the spaces have been used so that you can pick the method that you think is most appropriate for your own requirements. Spaces, as I have noted earlier, do not necessarily have to be put in, but inserting spaces makes sure that the string contains nothing that is unwanted.

Once again, the listing shows the header section as well, and you can see that a new constant LL has been added for the line length, and variable NEWTX is of type LINE. The procedure LEFT requires the parameters TEXT, the 80-character string, and the position integer S. Integer J is used as the position indicator for the characters in the arrays, and the action consists of copying characters from one array to the other until J=S, the specified number of characters. From this point on, another REPEAT loop pads NEWTX with spaces until it reaches its correct length. The procedure is used by typing LEFT with the correct parameters (remember that the string parameter must be of the correct length, 40 characters in this example) and then WRITE(NEWTX) to print the slice. Of course, if you only wanted to print it, you would normally have used WRITE(TEXT:field), but this line shows that the slice has been correctly taken. The RIGHT procedure carries out a similar copying action, except that the length of the TEXT string has to be found by using LEN. In addition, blanks are copied from the TEXT string into the NEWTX string. This could be a risky operation because if the TEXT string was a long one, there might not be enough blanks in the TEXT string to fill the NEWTX string. The only reason for using this method is to show how neatly it can be done, but the method that is used in LEFT, of adding blanks until the length is complete, is better. The formula L−S+J for the character in the TEXT file ensures that the first character is taken from the correct place in the array. For example, if you have requested RIGHT(TEXT,3), and the string contains a word of ten characters, then L will be 10, and S is 3. Since J=1 for the first pass through the loop, L−S+J gives 10−3+1, which is 8. This is the correct character number for the start of the three-character string, and by incrementing J we copy the characters correctly.

The equivalent of the BASIC MID$ action is carried out by the MID pro-

cedure in Figure 8.5. In this procedure, TEXT is the 80-character string which is to be sliced, and S is the position of the first character that you want to copy. The integer F is the number of characters that you want to copy. Some MID$ instructions in BASIC use this second number as another position number, but in this procedure it is used in the same way as in the MID$ of Commodore 64

```
 10  PROGRAM P8N5;
 20  CONST MAX=80;
 30        LL=40;
 40  TAB='        ';(*8 SPACES*)
 50  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 60     LINE=PACKED ARRAY[1..40]OF CHAR;
 70  VAR TEXT,SPACE:STRING;
 80     NEWTX:LINE;
 90  X,Y:INTEGER;
 100  PROCEDURE MID(TEXT:STRING;S,F:INTE
GER);
 110  VAR J,N:INTEGER;
 120  BEGIN J:=0;
 130  REPEAT J:=J+1;
 140    NEWTX[J]:=TEXT[S+J-1];
 150  UNTIL J=F;
 160  J:=J+1;
 170  FOR N:=J TO LL DO
 180  NEWTX[N]:=' ';
 190  END;
 200  BEGIN;(*MAIN PROGRAM*)
 210  PAGE;
 220  WRITELN('PLEASE INPUT YOUR TEXT');
 230  READLN(TEXT);
 240  WRITE('THE MID(3,6) IS ');
 250  MID(TEXT,3,6);
 260  WRITELN(NEWTX);
 280  END.
```

Figure 8.5 The **MID** procedure.

BASIC. The action is the same as has been used for LEFT and RIGHT, and it consists of copying characters from TEXT to NEWTX. The character position is obtained by using S+J−1, and when the selected number of characters has been copied (J=F), then NEWTX is filled with blanks using, this time, a FOR loop. The example program then prints the NEWTX string to prove that it has been correctly sliced.

Another useful function which is provided on many computers, but not in the BASIC of the C-64, is STRING$. This will make up a string of a number of identical characters, with the number and the character specified. For example, in Microsoft BASIC, X$=STRING$(20,"*") will make X$ a string of twenty asterisk signs. This is a quite simple action to achieve in Pascal. In addition, most varieties of BASIC have the STR$ function. This converts a number variable into a string variable; something that is quite often needed in BASIC. Pascal, with its strict rules about data types, is not provided with such trimmings, but it's easy to write a procedure which will convert a positive integer number into string form. It would not be too hard to develop from this a procedure which could cope with a reasonable range of real numbers, provided that they were always in ordinary form. You can, of course, enter numbers into a text-file, as when you use READ or READLN.

Figure 8.6 shows the STRING$ equivalent and the STR procedures. The variable STRN is of type NUMBER, a packed array of five characters. This will take the five figures of a positive integer, and it is the string that will be produced by the STR procedure. The FILL procedure is the one that emulates the STRING$ action. The parameters that have to be passed are the numbers of characters, and the ASCII code. As usual, the procedure counts in the number of characters into the array, and then fills with blanks so that the dimensioning of the array NEWTX is correct. This procedure is called by typing FILL, followed, in brackets, by the number of characters and the ASCII code for the characters. Note that only one character can be specified in one FILL instruction. The result of FILL is the string NEWTX filled with the characters. There is no check on the number that is passed to this routine, so that it would be possible to stop the program by specifying a number which was greater than the dimensioning of the array. Since these are skeleton procedures, you can fill in the fleshy bits for yourself.

The integer to string procedure needs more care. The restriction to positive integers makes the programming much easier, but there remains the problem that if the number is small, like 5, it will be printed with leading zeros, as 00005. This can be dealt with by a procedure which strips the zeros, or by a function which measures the number and returns a field number. The procedure starts by assigning the number 10000 to variable DV. This is the number which, when divided into a five-figure integer, will give a whole number result, and a four-figure remainder. The line J:=J DIV DV gives the whole-number result of division. For example, if J=12345, then J DIV DV gives 1 and J MOD DV would give 2345. We then take advantage of the fact that the ASCII code for a digit is 48 + the digit value. This is integer C, and it is placed in the string in line 270, using CHR(C) because a string must be packed with character variables. The remainder from the division is then assigned to J, and DV is divided by ten to act as the divisor for the next time round. Looping round five times will convert any positive integer into a five-character string, STRN, which can then be printed. Working with negative integers or with real numbers poses the prob-

```
 10  PROGRAM P8N6;
 20  CONST MAX=80;
 30        LL=40;
 40  TAB='        ';(*8 SPACES*)
 50  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 60     LINE=PACKED ARRAY[1..40]OF CHAR;
 70   NUMBER=PACKED ARRAY[1..5]OF CHAR;
 80  VAR TEXT,SPACE:STRING;
 90     NEWTX:LINE;
100   STRN:NUMBER;
110   X,J,C:INTEGER;
120   PROCEDURE FILL(J,C:INTEGER);
130   VAR N:INTEGER;
140   BEGIN;
150    FOR N:=1 TO J DO
160    NEWTX[N]:=CHR(C);
170    FOR N:=J+1 TO LL DO
180    NEWTX[N]:=' ';
190   END;
200   PROCEDURE STR(J:INTEGER);
210   VAR DV,N,K,C:INTEGER;
220   BEGIN;
230   DV:=10000;
240   FOR N:=1 TO 5 DO BEGIN
250    K:=J DIV DV;
260    C:=48+K;
270    STRN[N]:=CHR(C);
280    J:=J MOD DV;
290    DV:=DV DIV 10;
300    END;
310   END;
320   BEGIN;(*MAIN PROGRAM*)
330   X:=2345;
340   STR(X);
350   WRITELN(STRN);
360   WRITELN;
370   FILL(20,64);
380   WRITELN(NEWTX);
400   END.
```

Figure 8.6 Procedures **FILL** and **STR** to carry out the equivalents of **STRING$** and **STR$** in BASIC.

lems of recognising characters that are not digits, like the minus sign and the decimal point. This complicates the procedure considerably. The procedure is called as illustrated in lines 330 to 350.

The opposite effect from STR$ in BASIC is performed by VAL. In most varieties of BASIC, the VAL of a string which consists of digits is a number variable. For example, you can program V=VAL(A$) and if A$="123" then V has the value 123. As usual, this is a forbidden type of operation in Pascal, and we can perform it easily only for strings which represent positive integers, with a total of five characters. The procedure makes use of MID to extract the characters from the string, one by one, and the conversion is done by finding the ASCII code for the character and subtracting 48. This resulting number then has to be multiplied by the correct power of ten, and added to any previous results. The simplest system involves reading the digits starting at the right-hand of the string. This is illustrated in Figure 8.7, in which the new procedure is VAL. Line 250 sets values of M, the multiplier, and K which will be the total integer value of the end result. Note that this is a function rather than a procedure, since it gives a number result. By choosing to loop 5 DOWNTO 1, we pick characters starting from the right-hand side of string STRN. The MID action, which has had to be modified to take a five-character string, is then used to pick out one character. This gives the string NEWTX, and the character itself is NEWTX[1]. The ASCII code for this character is found in line 280, and by subtracting 48, we get the number itself. On the first pass through the loop, this is the units figure, and it is multiplied by one and added in to the total K. The multiplier M is then multiplied by ten, so that the tens figure which is extracted on the next pass can be correctly treated, and so on. The IF clause in line 310 is necessary because M would normally be multiplied by ten again on the last pass, making it 100 000. This is beyond the range of an integer, and would cause an error message.

CONCATENATION AND INSERTION

Concatenation means joining two strings together, and it is achieved in most dialects of BASIC by using a + sign between strings. There is no similar instruction in Oxford Pascal, though some versions of Pascal achieve concatenation of strings with the * sign. These are the versions which have the STRING type as a built-in data type. We can, however, easily write skeleton procedures which both concatenate and insert strings. Insertion in this sense means that one string is added to the other at some intermediate position. You might, for example, insert DIME into SENT so as to get SEDIMENT, though it's hardly likely to be something that you would do very often! Nevertheless, these are useful actions at times, and they can be provided with the procedures of Figure 8.8.

Unlike most of the procedures that we have used to date, these are required

```
 10  PROGRAM P8N7;
 20  CONST MAX=80;
 30          LL=40;
 40  TAB='        ';(*8 SPACES*)
 50  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 60     LINE=PACKED ARRAY[1..40]OF CHAR;
 70  NUMBER=PACKED ARRAY[1..5]OF CHAR;
 80  VAR TEXT,SPACE:STRING;
 90      STRN:NUMBER;
 100  NEWTX:LINE;
 110  X,Y:INTEGER;
 120  PROCEDURE MID(TEXT:NUMBER;S,F:INTE
GER);
 130  VAR J,N:INTEGER;
 140  BEGIN J:=0;
 150  REPEAT J:=J+1;
 160    NEWTX[J]:=TEXT[S+J-1];
 170  UNTIL J=F;
 180  J:=J+1;
 190  FOR N:=J TO LL DO
 200  NEWTX[N]:=' ';
 210  END;
 220  FUNCTION VAL(STRN:NUMBER):INTEGER;
 230  VAR C,K,N,M:INTEGER;
 240  BEGIN;
 250  M:=1;K:=0;
 260  FOR N:=5 DOWNTO 1 DO BEGIN
 270  MID(STRN,N,1);
 280    C:=ORD(NEWTX[1]);
 290    C:=C-48;
 300    K:=K+M*C;
 310    IF M<5000 THEN M:=M*10;
 320    END;
 330  VAL:=K
 340  END;
 350  BEGIN;(*MAIN PROGRAM*)
 360  PAGE;
 370  WRITELN('STRN IS 12345');
 380  STRN:='12345';
 390  WRITE('THIS BECOMES INTEGER VAL');
 400  WRITELN(VAL(STRN));
 420  END.
```

Figure 8.7 Function **VAL** in Pascal, but for integers only.

```
 10  PROGRAM P8N8;
 20  CONST MAX=80;
 30        LL=40;
 40  TAB='        ';(*8 SPACES*)
 50  TYPE STRING=PACKED ARRAY[1..MAX]OF
CHAR;
 60     LINE=PACKED ARRAY[1..40]OF CHAR;
 70  VAR TEXT,SPACE:STRING;
 80     STR1,STR2,TMP1,TMP2,CCAT:LINE;
 90  X,Y:INTEGER;
 100  FUNCTION LEN(S:LINE):INTEGER;
 110  VAR J:INTEGER;
 120   BEGIN;
 130   J:=LL;
 140   WHILE S[J]=' 'DO J:=J-1;
 150  LEN:=J;
 160  END;
 170  PROCEDURE CONCAT(STR1,STR2:LINE;VA
R CCAT:LINE);
 180  VAR L,M,J:INTEGER;
 190  BEGIN;
 200   J:=0;
 210   L:=LEN(STR1);
 220   M:=LEN(STR2);
 230   REPEAT
 240    J:=J+1;
 250    STR1[L+J]:=STR2[J];
 260    UNTIL (J=M) OR (L+J=LL);
 270   J:=J+L;
 280  CCAT:=STR1;
 290  END;
 300  PROCEDURE INSERT(STR1,STR2:LINE;J:
INTEGER;VAR CCAT:LINE);
 310  VAR L,M,N,K:INTEGER;
 320  TEMP:LINE;
 330  BEGIN;
 340   L:=LEN(STR1);
 350   M:=LEN(STR2);
 360   N:=0;K:=J;
 370   REPEAT;
 380    N:=N+1;K:=K+1;
 390    TEMP[N]:=STR1[K];
 400   UNTIL N=L-J;
 410   K:=0;
```

```
420    REPEAT;
430     K:=K+1;
440     STR1[J+K]:=STR2[K];
450    UNTIL (K=M) OR (J+K=LL);
460    J:=J+M;K:=0;
470    IF J<LL THEN  BEGIN
480     REPEAT
490     J:=J+1;K:=K+1;
500     STR1[J]:=TEMP[K];
510     UNTIL (K=L-J) OR (J=LL);
520    END;
530    CCAT:=STR1;
540  END;
550  BEGIN; (*MAIN PROGRAM*);
560  PAGE;
570  WRITELN('FIRST STRING');
580  READLN (STR1);
590  WRITELN('SECOND STRING');
600  READLN(STR2);
610  X:=ROUND((LEN(STR1))/2);
620  CONCAT(STR1,STR2,CCAT);
630  WRITELN('CONCAT GIVES ',CCAT);
640  INSERT(STR1,STR2,X,CCAT);
650  WRITELN('INSERT GIVES ',CCAT);
670  END.
```

Figure 8.8 String concatenation and insertion procedures.

to return a new variable. This variable has been called CCAT, but any declared variable of the correct type (LINE) could be used when the procedure is called. Up to now, our procedures have printed any quantity that was required as the end result, and any attempt to use outside the procedure any of the variables that appears in the procedure is doomed to failure because all parameters in a procedure are local. For example, if the strings that are passed to a procedure are STR1 and STR2, and the procedure then alters STR1, then this altered value is purely local. Printing the value of STR1 after the procedure has run gives the same value as before the procedure started. If you want to get a value like this back from a procedure, you must declare it as a variable that you want to be passed back. This has its advantages, because it means that using a procedure with a variable quantity passed to it will not alter the value of the variable. This saves a lot of temporary assignments, with all the hassle of additional declaration that it would involve. When you want a procedure to come back with a result in the form of a variable, you have to specify the variable both in

the call to the procedure and in the procedure header. The addition to the calling instruction is simply to add the new variable name at the end. For example, if you want to use STR1 and STR2 and come back with STR3, then your call would be CONCAT(STR1,STR2,STR3), using the CONCAT procedure as an example.

The change to the header of the procedure is more complicated. After the usual declaration of the variables that are passed to the procedure, there is a semicolon, and then something like VAR STR3:LINE. This looks like a normal variable declaration, but is part of the procedure header. The variable which is named in this section will be available for passing back data. This does not mean that you must use the same variable name in your main program, but it does mean that you must include a variable of the same type and in the same position in your procedure call. Remember that for two strings to be of the same type, they must both consist of packed arrays of the same number of characters. In the example of Figure 8.8, both procedures have been arranged to pass the result to a variable CCAT, and to avoid having to declare another variable name, this has also been used in calling the procedure and afterwards in printing the result.

The CONCAT procedure starts in line 170. The variables that are passed to the procedure are two strings of 40-character length, and the variable CCAT will be assigned with the concatenated string. You could, if you liked, alter the procedure so that CONCAT was an 80-character string. This would avoid any possibility of the concatenation action producing a string that was too long, and so stopping the run with an array error. Variables L, M and J are integers, and at the start of the program, J is equated to zero, and L, M are the lengths of STR1 and STR2 respectively. The function LEN must exist in the program in order to carry out this determination of length. The first loop in lines 230 to 260 then adds the characters of STR2 into STR1, using variable J as an index for STR2 and L+J for STR1. This has the 'safety-net' step of stopping the loop if the value of L+J reaches the maximum allowed for this string type. The concatenated string STR1 is then assigned to CCAT. When the procedure has ended, CCAT will have the value of the concatenated string, but STR1 will still have its original value.

The INSERT routine is rather more complicated. The action is to copy into a temporary variable the second half of the string that is to be split. The second string is then concatenated to the first part of the first string, and the temporary variable is added back. The procedure requires three variables to be passed to it. Two are the strings that have to be used, and the third is the position in the first string following which the second string will be inserted. There must be a string to return, and this is declared as a variable CCAT in the heading in the usual way. The integers L and M are assigned with the lengths of the strings, using LEN as before, and variables N and K are used as counters. K is assigned with the value of J, the 'break-in' point in the first string.

Lines 370 to 400 then assign to the string TEMP, the remainder of STR1 from

character K+1. This is the character which follows the one at which the break is to be made. Since the break is made after character J, and L is the length of the string, the rest of the useful strain must consist of L−J characters. The next action has to be to concatenate all of the characters of the second string on to the first from the break point. This could be done by calling CONCAT, but I have avoided this in case you needed only the one routine. The concatenation is done in lines 420 to 450, and the ending condition has been phrased so as to stop the action if STR1 reaches its last character. Similarly, the re-joining of the broken part of the first string is carried out only if there is room to do so by making the action conditional in line 470. The last loop copies the characters from TMP into STR1 for as long as there is room for them. Finally, the STR1 value is assigned to CCAT so that the value can be passed back.

Note that in these actions there is no need to pad strings with blanks, because each string consists of an array of fixed length, 80 characters in this example, which has been assigned previously. Where we talk about 'string length', we mean the useful characters of the array. This number can vary, depending on how the array has been assigned, but the total lengths are constant and are automatically padded with blank spaces by the READLN action in the main part of the program.

9

ASSORTMENT

In a book of this size, it's impossible to cover all the possibilities that a language like Pascal offers, and I have aimed at dealing with the essential features that most users of Oxford Pascal on the Commodore 64 will need. Nevertheless, we have covered most of the important topics in reasonable detail, and the main thing that you will need from now on is practice. Only a lot of practice, particularly in planning programs, will release you from the frustration of finding errors reported each time you compile, and then getting a different set of errors when you try to run your program. In this chapter, we shall look at some points that have been skipped over previously, or which need more emphasis. The most important of these topics is fault finding.

Inevitably, when you start to learn a new language, you are going to make a lot of errors in syntax. This is particularly true if you have previously programmed only in BASIC. The most common mistakes are omitting semicolons, and forgetting the brackets and the *single* inverted commas in WRITE/WRITELN statements. Many of these errors are found by the compiler and reported in such a way that they should not be difficult to remedy. There are several such syntax errors, however, which produce reports that don't lead you to the fault unless you have had some experience. If, for example, you revert to BASIC habits and write an array member with round brackets instead of square brackets, you can get some very interesting error reports. For example, if you have the line

```
STR(J):=S[J];
```

then you can get the report

```
':=' EXPECTED
```

followed by

```
TYPE MISMATCH IN ASSIGNMENT STATEMENT.
```

which will certainly draw your attention to the line that causes the problem, but don't remind you of what the fault is. If the brackets error is in a WHILE loop, such as

```
WHILE S(J)=' '
```

then you may get the error message

```
BOOLEAN EXPRESSION REQUIRED AFTER WHILE
```

meaning that S(J)=' ' cannot be true or false because S(J) has no meaning at all. Another fruitful source of error messages lies in assignments. You become so accustomed to the few data types in BASIC, that you assume that almost any variable can be equated to any other. In Pascal, data types are treated very strictly, and you can't even assign a variable which is defined as an 80-character array to one which is a 79-character array. This is probably one of the errors that causes most annoyance to ex-BASIC programmers.

In addition, even the most obvious error messages may not be all that they seem. You will often get a message about a missing terminator when you can see from the listing that each line has its correct semicolon. Apart from the fact that a missing semicolon is sometimes not detected in the line at which it occurs, there are other error conditions that can give this message. One common one is writing

```
IF X=Y THEN DO REPEAT
```

rather than

```
IF X=Y THEN DO BEGIN
REPEAT;
```

and this can also cause other errors, like a message to the effect that := is needed. This usually is issued just following where the := appears in the program and for anyone brought up on BASIC error messages it can raise your blood pressure to an unhealthy extent. You can even get missing terminator reports because you have put in a terminator where there should not be one. The most important rules concerning semicolons are that there should be none following DO, and none following a remark which is the first line of a program. If you have a first line like

```
5 (*PROGRAM WRITERIGHT*);
```

then the compiler will issue a 'BEGIN' EXPECTED error message on this first line, and probably a MISSING TERMINATOR message on the next line. Many BASIC programmers are accustomed to starting a program with a REM line, and it can be difficult to break the habit. If you make use of remarks, either write them following a statement which is correctly terminated, or put them in a line of their own with no semicolon. A semicolon is acceptable if the REM line is not the first line, but it makes life easier if you can keep to one method. It's better, in fact, to get into the habit of starting a program with a PROGRAM statement which must end with a semicolon.

Another potent source of error messages is forgetting a final apostrophe in a WRITE or WRITELN statement. This can be picked up many lines later and result in a message such as END EXPECTED. The reason is that the compiler has taken everything following the first apostrophe as something to be printed, and has not correctly read an END. It's equally easy to skip a BEGIN this way, and get a message to the effect that END; should be END., when you know that

the program should not have ended at this point. It pays to be particularly careful about WRITE statements, because omitting the apostrophe can cause an error message that is so remote from the line that caused it that the source is very difficult to trace.

Even when a program compiles correctly, there is no guarantee that the program will run without fault. An error-free compilation means only that the statements in the program are of correct syntax and do not contain any undeclared variables or impossible assignments. You can compile without errors, for example, a program that will try to put more items into an array than the array has space for, or which has a faulty CASE statement in which you can enter a reply for which no CASE is provided. In general, run-time errors are caused by faulty planning of one sort or another, rather than by faulty typing or bad use of statements. You may have omitted to test the size of an entered quantity, for example, or given no thought to how many items could be put into an array. If you are experienced in programming with any other language, then the run-time errors should present no problems to you.

HINTS AND TIPS

There are a few odd hints and tips which can make your Pascal programming career rather easier. One concerns the renumbering of programs using NUMBER. You will find that when a source program is loaded from a disk, the line numbering starts at 1000. This can often be inconvenient, and you try to renumber it with the NUMBER command. This will cause problems, however, because NUMBER will not renumber a program with a number for the starting line that is less than the existing number.You can't, for example, type NUMBER1000,10,10 so as to make the program that starts at line 1000 become renumbered to one which starts at line 10. The way round this is to put in a dummy line 5, such as 5 (**) and then do NUMBER5,5,5. Now delete line 5 by typing 5 and pressing RETURN, and renumber with NUMBER10,10,10. This will renumber your program in tens, starting with line 10. The NUMBER command is also very useful when you want to make room for another procedure or some new lines in a program. If you select the line following the one at which you want to start inserting, you can number all the following lines higher. This leaves room for new lines with no risk of removing existing lines. For example, if you want to insert a new procedure after line 120, you can type NUMBER130,1000,10 to make the next line number equal to 1000. This gives you lines 130 to 990 to play with, and if you don't use them all, you can get back to normal numbering by using NUMBER10,10,1.

The editing system of the Commodore 64 itself, along with the automatic numbering system of Oxford Pascal can cause a number of problems until you become accustomed to the way that it works. When you want to edit a line, you will normally either list that line alone, or list it along with the surrounding

lines. Unless you have a printer and have made a paper copy of a program that is causing bother, you will probably want to list the lines that surround the faulty line. Now when you correct the fault in the usual way and press RETURN, the cursor will move to the next line in the usual way. What is less usual, if you have programmed only with Commodore 64 BASIC in the past, is that the line may have been renumbered! Suppose, for example, that you happen to be working on line 400 and the next line, because of some insertions, is line 405. Now if you edit line 400, correcting a square bracket to a round bracket perhaps, and then press RETURN, the next line number will appear to be 410, unless you have previously changed to numbering in fives. At this point, if you press RETURN, two things will happen. One is that if there was a genuine line 410, it will be replaced by another copy of line 405. The other is that you now have two lines, 405 and 410, which are identical, and you will have lost the old line 410. A better response is *always* to press the down-cursor key rather than RETURN immediately after you have pressed RETURN to complete editing. The same sort of thing can happen if you are editing a line which is displayed on its own. If you edit 400 and press RETURN, the line number 410 will appear. Press RETURN on this, and you will have totally erased any line 400 from your program. On the whole, because of this, it's better to work on lines in groups because you will then have a copy of a line if anything untoward happens. Get into the habit when you have finished editing a line, of pressing RETURN, and then using the down cursor key to take the cursor clear of all the lines until you can decide what to do next.

If you have not had much experience of the Commodore 64 editing system with BASIC, there are other traps for you. The important thing to remember is that what appears on the screen will be stored in memory when you edit. You must, for example, avoid carrying out editing on a set of lines which are left on the screen during compiling. When an error appears on compiling, it's better to deal with it at once, rather than wait for others to appear. This is because you may find that removing the first error will also remove the others. It makes sense, then, to use the STOP key to stop compiling, and then to edit out the mistake. If you do this straight away, though, you may find that you have edited in your error report that the compiler delivered!. Always LIST the lines that you want to edit, rather than deal with them with the error message showing, and always LIST edited lines after editing is complete, in case anything has strayed. I once spent an hour looking for a fault which turned out to be a stray apostrophe which had been on the screen during editing and had become placed in a blank space in a line by an editing action.

Sometimes, of course, all this can be turned to good use. Suppose, for example, that you want to make a copy of a complete routine and place it elsewhere in a program. If you design your programs correctly, of course, you should not need to do this very often, but the need does arise at times. The method is to place on the screen, using LIST, the lines that you want to copy. Make sure that they are numbered in tens, or in whatever increment you may

have used in the AUTO command previously. Now edit the line number of each line. This involves changing the first line number, and from then on, just pressing RETURN on each line until you get to the last one in the group. Suppose, for example, that you had lines 900 to 950, and you changed the first line number to 200. The result of pressing RETURN on each line would be to show on the screen lines numbered 200 to 250. When you press RETURN for the last time, you will have two sets of lines. One set, numbered 900 to 950, will still be held in the memory, and the new set of 200 to 250 will also be in the memory. If you want to remove the old set, you can use DELETE 900–950 (RETURN). You can carry out this exercise as many times as you want if you need several copies for any reason. You must be careful when you create line copies that the copy does not replace some other line that will be needed. In general, if you want to work on long programs, you will find that a printer is an essential rather than a luxury, because when you can look at the whole of a program, or at least a very substantial section of it, at one glance, it's very much easier to see what needs to be done.

Finally, don't forget the very useful find-and-replace facilities of the Oxford Pascal editor. As always, if you have programmed only the Commodore 64 in BASIC you will not have come across these useful aids to programming. If you have used any good word-processor software, however, you will know how much time can be saved by using search-and-replace actions. In a program which consists of a large number of WRITE and WRITELN statements, for example, you can save a lot of typing time by abbreviating WRITE(' to W. and WRITELN(' to WL. You can make similar economies with READ and READLN. If you use a lot of these search-and-replace actions, though, it's a good idea to keep a note of which abbreviation you have used for which purpose. If you don't, there is a risk of using the same abbreviation for two different instructions, or of forgetting that you have used an abbreviation. These editing actions are particularly useful, of course, in versions of Pascal which do not use line numbers, but if you do not have a printer, it's very handy to be able to find, for example, which line contains something like (*FIND DATA*).

DISK MODE OPERATION

This book ends with a brief description of the additional facilities that are available in disk mode. The description is brief because, as I have explained, disk mode is useful only when your grasp of Oxford Pascal is good enough to encourage you to write long programs that make full use of the disk system, or which will be used as 'stand-alone' programs. A 'stand-alone' program in this sense means one that is in machine code that will execute on any C-64, whether it is equipped with Oxford Pascal or not. Obviously, if you use only resident mode Pascal from cassette, this section will be of no interest, except as a

promise of delights to come when you buy a disk system. If you use disks already, this section shows how you can extend your use of Pascal considerably by taking advantage of disk mode.

TEXT FILES

We have previously covered the method that is used to compile a program in disk mode, and we shall not look at that again. What we need to concentrate on here are statements which work only when disk mode is in use. For the C-64, the most important of these are the statements that relate to textfiles. There is a special structured data type, called FILE, which is available in Pascal. This is another of the types which in BASIC you have to construct for yourself, but which in Pascal is ready-made for you. At the moment, we'll confine ourselves to just one type of FILE, the text file. On some versions of Oxford Pascal, this is the only type of file that the resident compiler can deal with, but on the Commodore 64 some text file actions are usable only in disk mode.

In BASIC, a file is something that you use only when you are saving data on cassette or disk, or recovering such data. Pascal deals with files quite differently. Files can be internal or external, and these descriptions tell you all. An internal file takes an input from the keyboard, and its output is the screen, or possibly the printer, with the file data kept temporarily in memory. An external file is stored on the disk, and its input and output processes are disk reading and writing respectively. Disk files are not handled by the resident mode. One important type of file which can be used internally or externally is the text file, in which each character is an ASCII code. The file in memory is stored like an array of CHAR, and the main difference as far as we are concerned is that the file does not have to be of a preset length. Each time you have used READ and WRITE, in fact, you have been dealing with a text file. Even if what you read was a set of numbers, these numbers were processed in ASCII form so that they could be seen on the screen or sent to the printer. If we want to be able to make more use of these files, however, we need to be able to write them to and read them from disk.

Figure 9.1 shows an example of a text file in use. The idea is to read five phrases, each of up to 40 characters, from the keyboard. This 'internal file' will then be recorded by tying it to an external file. This linking of the files is done by the REWRITE instruction, which needs to have both the internal and the external file names. The external file name must carry the drive number. The writing part of the program consists of lines 60 to 120. The REWRITE instruction contains the name TXT of the internal file, the phrases that you will type at the keyboard when the program runs. It also contains the name '0:MYTEXT' of the external disk file, the file that will be stored on the disk. The loop that starts in line 70 then reads phrases which you type at the keyboard, and transfers them to a disk file. This file is not recorded while you are typing, however. Disk

```
10  PROGRAM P9N1;
20  VAR N:INTEGER;
30  WORDS:PACKED ARRAY[1..40] OF CHAR;
40  TXT:TEXT;
50  BEGIN;
60   REWRITE(TXT,'0:MYTEXT');
70   FOR N:=1 TO 5 DO
80   BEGIN;
90    WRITELN('PHRASE, PLEASE');
100    READLN(WORDS);
110    WRITELN(TXT,WORDS);
120   END;
130  PAGE;
140  WRITELN('PRESS ANY KEY');
150  WHILE GETKEY=CHR(0) DO;
160   RESET(TXT,'0:MYTEXT');
170   FOR N:=1 TO 5 DO
180    BEGIN;
190      READLN(TXT,WORDS);
200      WRITELN(WORDS);
210    END;
230  END.
```

Figure 9.1 A text file being used to obtain quantities from the keyboard and store them on the disk.

operation is never like this, as you probably know. Instead, the characters are gathered up in a memory buffer ready to write on to the disk when one of a number of things happens. One of these can be the CLOSE instruction. Another can be a full buffer, but this isn't likely when you are typing only five phrases of up to 40 characters each. The last possibility is a REWRITE or RESET. In this example, then, you will not hear any disk activity after you have typed five phrases. When you 'press any key', however, the RESET in line 160 will cause the buffer to be emptied, writing your text to the disk and also preparing for a read. The read action in line 170 to 210 then takes place, placing your phrases on the screen. You will see that the line spacing is doubled, because each line was written on to the disk with a line feed, and has been read back with another line feed added because of the use of WRITELN as well as READLN.

Remember that the procedure for dealing with this program is different when you are using disk mode. You must insert the Pascal system disk (this is a case where you really must have a back-up), and type COMP P9N1,L. This assumes that you have used the filename of P9N1 for this example, but obviously you will type here whatever filename you have used. The L part of the

instruction ensures that a listing will be shown on the screen. When you press RETURN on this, the system disk will spin for some time, and you will then be prompted to remove the system disk and put in the disk which carries the program. When you do this, and press RETURN again, the program is read and compiled. If you get error messages in the course of this compilation, you can stop the compilation and change the program as you need, but you must remember to save the program *with the same filename* after correction. You will then have to go over the whole COMP procedure again, which is a considerable incentive to get your programs correct first time! When the compilation reports zero errors, you can run the program by typing EX P9N1, with the system disk in place. Once again, the system disk will spin, and you will be prompted to put in the disk that contains your program. The program will then run, and at the end of the run, you will be prompted to replace the system disk again. For these short examples, this is rather a long-winded way of using the system, but it has considerable advantages for long programs, not least that it makes it possible to create a very long program out of several sections.

The example has shown phrases being read with a numbered loop, and with the number of characters per line limited by the use of an array of type CHAR. A text file can, however, be used in a more 'free-range' form by dealing with a character at a time. This is illustrated in Figures 9.2 and 9.3, in which characters

```
10  PROGRAM P9N2;
20  VAR CH:CHAR;
30   TXFIL:TEXT;
40  BEGIN;
50  REWRITE(TXFIL,'0:MYFIL');
60   REPEAT
70    BEGIN;
80    WRITELN('DATA PLEASE');
90     REPEAT
100      BEGIN;
110      READ(CH);
120      WRITE(CH);
130      WRITE(TXFIL,CH);
140      END;
150     UNTIL (CH=CHR(13)) OR (CH='-');
160    WRITELN(TXFIL);
170    END;
180  UNTIL CH='-';
190  CLOSE(TXFIL);
210  END.
```

Figure 9.2 Working with a text file one character at a time. This is normally much more useful, because each character can be tested.

are read from the keyboard, echoed on the screen, and stored in a text file on disk. When you want to do this, you have to put in instructions that will mark the end of a line, and which will also mark the end of the file. In this case, the conventional CHR(13), the RETURN key, is taken as the end of line marker, but the end of the file is marked for the purpose of entry by pressing the C= key and the Z key together. The effect is that you can type as you please, using RETURN to terminate each line. When you have finished typing data, you can use C= and Z to end the file and record it on disk. The CLOSE(TXTFIL) then ensures that the file is closed, otherwise you will encounter trouble when you try to read the disk. CLOSE is an extension to standard Pascal which is peculiar to Oxford Pascal. A feature to note is that the characters appear on the screen as you type them (because of the WRITE(CH) step) and also in a complete group after each RETURN (because of the WRITE(TXFIL,CH) step). The character that should appear between the apostrophe signs in lines 150 and 180 is the graphics sign that is obtained by pressing the C= key and the Z key at the same time. On the listing, this appears as a horizontal bar, because of the use of an Epson printer rather than the Commodore printer.

OTHER DISK FILES

Though text files can be used for any quantities that you may want to store on disk, this is not always useful. If all that you are likely to do with file information is to print it, of course, text files are all that you might need. When you put numbers into text files, they will be printed with field widths which depend on the number type, and your manual for Oxford Pascal lists these field widths in the section that deals with text files. You will need to change these field widths with the appropriate number (following a colon) if you want to use anything different. For many record-keeping applications, the use of text files, with appropriate fielding for characters that represent numbers, is often perfectly satisfactory.

You can, however, create files of anything you like. You can declare that NUMFIL is, for example, a file of integer, or a file of real. You can also declare that GOLFCLUB is a RECORD, and then use GOLFIL as a FILE OF GOLFCLUB. In other words, for whatever types of variables your program might need to use, you can create a disk file of these types. Since Oxford Pascal allows you to use up to five sequential files at a time, there is scope for quite elaborate record-keeping. If you want to use the Commodore 64 facility for relative files, you will have to do this with the syntax that we looked at earlier in Chapter 4. Standard Pascal has no provision for using random access files, nor does Oxford Pascal. The lack of random access files in Standard Pascal was a feature that prevented the language from gaining converts for business use, but many dialects of Pascal, notably the Pascal for the IBM PC, have random access files available.

```
10  PROGRAM P9N3;
20  VAR CH:CHAR;
30   TXFIL:TEXT;
40   BEGIN;
50   RESET(TXFIL,'0:MYFIL');
60   WHILE NOT EOF(TXFIL) DO
70    BEGIN;
80    WHILE NOT EOLN(TXFIL) DO
90     BEGIN;
100    READ(TXFIL,CH);
110    WRITE(CH);
120    END;
130   WRITELN;
140   END;
160 END.
```

Figure 9.3 Replaying a text file that has been recorded by using the program of Figure 9.2.

LINKING, CHAINING AND INCLUSIVE COMPILING

Disk mode compiling allows you some valuable facilities which will be of considerable use to you when your use of Pascal develops to the stage at which you need to write really long programs. By that time, you will probably have a disk that is almost filled with procedures and other program segments which you will find useful in programs. These need not have been created with the compiler in disk mode, but when you make use of disk mode you can carry out actions which are known as linking, chaining and inclusive compiling, all of which make it much easier to create long programs. The only snag is that you must know and understand the rules by which these actions operate.

We'll start with linking. Linking means that several object code sections can be combined into one program. This isn't just a matter of joining one program to another, and it can take quite a lot of time to complete. Each section must be a compiled program, one which has the .OBJ ending to its filename on disk. The .OBJ ending does not need to be placed into the filenames when the linking command is formed, however. If you have the sections FIRST.OBJ and SECOND.OBJ, then the two can be linked into one object code program with the file name of FINAL.OBJ by the command

```
LINK 0:FINAL=FIRST,SECOND
```

which is issued with the system disk in place—you will be prompted from then on when to insert the disk that contains the program files. When the linking is complete, the file FINAL.OBJ on the program disk will be the linked files.

Linking is not, however, quite so straightforward as it sounds. The programs

may link, but they will not run correctly unless you have obeyed the rules, which are straightforward but strict. The first rule is that the first of the programs that are to be linked must contain the main program, and the rest should contain only functions and procedures. Each of these secondary programs must then contain a dummy main program, consisting of the instructions BEGIN and END. In addition, there are strict rules about the main (outer-level) procedures and functions, meaning those which are called directly from the main program. These main procedures and functions must be contained in one section only, preferably (because it makes life easier), the first section that also contains the main program. The other sections can then consist of other procedures and functions. What you must avoid is having a procedure or a function declared twice. In the course of the program, if a procedure is to be used that is stored in another section, it has to have a 'dummy' header with the word EXTERN added. If, for example, you want to use procedure WORDCOUNT, and this is part of another section, you should 'define' the procedure by the line

```
PROCEDURE WORDCOUNT;EXTERN;
```

and *not* by repeating the steps of the procedure. The procedure can, of course, be called simply by using its name in the usual way. Finally, all the sections that are to be linked must have identical variable declarations for the main program. The variables that are used in procedures and in functions are local only, and do not have to be identical.

Because of these restrictions, you can't simply have a set of procedures stored on disk in object code form ready to use. You can, however, have a set of source code programs which need only alteration to variable declarations and to procedures to be compiled and then linked. Because of the search-and-replace facilities of Oxford Pascal, this type of modification is not difficult. Unless the sections contain disk instructions, they can normally be compiled and checked in resident mode at first, making life considerably easier. It pays, in fact, to have one section of source code with all your disk filing instructions on it, and to adapt this as you need it. Only this particular section will then need to be compiled in disk mode when the sections are tested. Once all of the sections have been tested, they are compiled into object code, using disk mode and the sections can be linked together.

Chaining is a rather different technique. When you link sections together, you end up with a longer program which can then be run. When you use chaining, you command one program to run another, but this does not make the first program longer, because the program that is called exists only for the time when it is running. Suppose, for example, that you had a program called MAINONE with two sections called ERRORMES and PRINTRESULTS. Your main program might be concerned with processing data, and would call up the section ERRORMES only when the main program needed to print a screenful of instructions to deal with an error. In the same way, the routines in PRINTRESULTS would be called up when the printing of the results of the main program was needed. This type of action becomes very useful for running very

large programs, because large procedures can be kept quite separately on the disk and called in only when they are needed. As you might expect, there are rules and restrictions. The size of the main program plus the longest section that it can chain must not exceed the memory space. If the largest section proves to be too long, then it too will have to be broken down into more usable sections. The total length of the main program plus all of its sections can, however, be as long as the disk capacity permits. If you have two drives, the total length does not even have to be restricted by the amount that can be placed on one disk! Any variables that are to be global, used by all sections, must be declared in all sections. If the main program has left any files unclosed when a section is chained, then the files will automatically be closed. This means that if the chained section needs to use the files, they will have to be opened again. A section is chained by having as part of the main program the line

```
CHAIN('SECTION1 ')
```

—using the correct filename of the object code file, which in this example would be SECTION1.OBJ on the disk. You can use a variable name (a packed array of CHAR) in place of the name between apostrophes, but whatever name you use in this way *must end with a space.*

Inclusive compiling is the third technique for using programs in the form of sections. Unlike linking and chaining, inclusive compiling operates on source code rather than on object code files. According to the manual, when a line of source code consists of the hash mark (#) followed by a filename, the source code of that filename will be compiled at that point and will become part of the program that is being compiled. As the manual points out, if you are assembling a number of sections that are to be chained or linked, then the global declarations can be made into a separate file, and compiled into each separate section in this way. I could not find an acceptable syntax for this instruction, however. No matter how I tried the action, the compilation always stopped with a 'File not found' error message. This may have been a bug in the version of the program which I had, but it does mean that you should check out this facility with unimportant examples before you attempt to use it for anything important.

TAIL-ENDERS

There are a few topics which simply don't fit into any classification, and which have to be grouped into a bundle like this of unrelated items. One of these is the use of GOTO. Now GOTO in BASIC is the command which makes the simpler versions of BASIC (like Commodore 64 BASIC) useable. It is, however, an instruction which can make BASIC programs almost unreadable, and often just about unworkable. The snag with GOTO in BASIC is that it causes a jump to some numbered line, and since each line has its own number, you can make a program jump anywhere, either by intent or by accident. If a BASIC program

is badly planned and uses a lot of GOTO statements, it can be very difficult to work out what is happening in the program. It is even more difficult to work out what is wrong when the program misbehaves.

In general, the use of three different types of loops in Pascal should mean that GOTO is almost never needed, and you'll see that none of the examples in this book makes use of GOTO. If you *must* use a GOTO, however, because it would simplify the programming of some loop condition, then Oxford Pascal, like standard Pascal, provides for the use of GOTO. The difference is that because the line numbers that you see on the screen are purely temporary, you have to mark the place to go with a 'label'. Since we're dealing with Pascal, we also have to declare the label. A label can be any integer number, but you must not attempt to use the same number to label two different places in a program. The label is declared in a line that starts with the word LABEL and is followed by a list of the label numbers, like LABEL 1,2,3,4; and only the label numbers that have been declared in this way can be used. To label an instruction, you start it with the label number, then a colon. Each GOTO that you use must then be followed by a label number. If a label has not been declared, you will get an error signalled when you compile. If you declare a label, but use it incorrectly by using, for example, GOTO 3 when no statement starts with 3:, then the error will be picked up only when the program runs.

Figure 9.4 shows a simple example of GOTO in use with a label. The label is put in front of a statement that is to be repeated. The label has been declared, and the GOTO in line 50 will therefore cause the WRITELN statement to repeat until a key is pressed. This action could, however, have been carried out just as easily by using a REPEAT...UNTIL loop. A GOTO is more likely to be useful when you want to break out of a loop by testing some condition in the middle of the loop. The WHILE type of loop tests at the start, and the REPEAT loop tests at the end, but neither makes any allowance for testing in the middle. If you must do this, then, GOTO offers a way. I must stress, though, that it's unusual to find any action in which the use of GOTO is completely unavoidable.

```
10  PROGRAM P9N4;
20  LABEL 1;
30  BEGIN;
40  1:WRITELN('REPEAT THIS');
50  IF GETKEY=CHR(0)THEN GOTO 1
70  END.
```

Figure 9.4 Using **GOTO** with a label number. The main application is for jumping out of the middle of a loop with an **IF** test.

FORWARD REFERENCES

Throughout this book, I have stressed the point that everything has to be arranged in the order that the compiler needs for efficient compilation. What

this boils down to is that no identifier should be used before it has been defined. There is, in fact, a minor exception to this principle. You may, in the course of a procedure, need to call another procedure. If this new procedure has not been defined, it can be declared with the word FORWARD in the definition to show that the definition will follow. Suppose, for example, that you need to call a procedure PRINTDATA from another procedure. You would need to have put in the section that deals with procedure definitions the line

```
PROCEDURE PRINTDATA(parameters);FORWARD
```

giving the name and the parameter list as usual, but with FORWARD instead of the full set of lines of the procedure. You can then place the actual procedure elsewhere, using only PROCEDURE PRINTDATA as the header. This type of FORWARD reference is not really needed in Oxford Pascal because of the excellent editing facilities of the C-64, but has been included because it is a part of standard Pascal.

OTHER INSTRUCTIONS

MAXINT is the number 32767, meaning the maximum number that can be represented as an integer. You can think of this as being a pre-defined identifier, and it can be used in tests for number size. Another number function is ODD. This operates on integers and returns a TRUE value if the number is odd. You can, for example, have a line such as

```
IF ODD(DATA) THEN MAKEVEN;
```

meaning that if the integer DATA is an odd number, then procedure MAKEVEN is to be carried out.

Of interest mainly to machine code programmers, numbers can be expressed in hex by using the dollar sign, so that $C000 is another way of writing the number 49152. You can read and write in hex numbers by using RDHEX in place of READ and WRHEX or WRHEX2 in place of WRITE. These are ready-made procedures and need to be called with the correct parameters. WRHEX2 is provided so that a single byte number can be printed as two hex digits, and for four digits WRHEX can be used. RDHEX will read up to four hex digits. The syntax of these instructions is demonstrated in the manual. You can also manipulate integers in Oxford Pascal by using ANDB, ORB, XORB, SHL and SHR in the way that you can use AND, OR, XOR, ASL and LSR in assembly language.

If you have routines in machine code which you can run from BASIC by using SYS, then you can run them from Pascal by using EXTERN. This assumes that the machine code can be placed in some part of the memory that is not used by Pascal. This, in practice, is not particularly easy. The machine code is called as if it were a function or a procedure, and it must include a normal function or procedure header. The action of the function or procedure is replaced by EXTERN, which must be followed by the starting address of the machine code.

Calling the function or procedure will then activate the machine code. Any values that have to be passed to or from the machine code routine are passed on the stack.

OTHER ACTIONS

The entry error messages of Pascal, which can result in a program stopping when an incorrect value is entered, can be disabled by using IOTRAP(FALSE). This will prevent a typing error from stopping a program with an error message, as Figure 9.5 shows. This example calls for an integer entry. With normal trapping present, entering a letter at the input stage will cause the program to stop

```
10  PROGRAM P9N5;
20  VAR N:INTEGER;
30  BEGIN;
40  WRITELN('NUMBER, PLEASE');
50  IOTRAP(FALSE);
60  READLN(N);
70  WRITELN(N);
80  IOTRAP(TRUE);
100  END.
```

Figure 9.5 Using **IOTRAP(FALSE)** to prevent a program from hanging up with an error message during entry of data.

with an INTEGER ENTRY ERROR message. With IOTRAP(FALSE), however, there is no error message. The manual states that the type of error can be analysed with IOERROR, but I was unable to make this work, and IOERROR seemed to remain at the value of zero no matter what was entered. This is a pity, because the use of IOERROR in a REPEAT loop (or even with GOTO) would be a useful way of making an entry section repeat until an acceptable entry was received. Another command which did not work in my version of Oxford Pascal was RESTORE. This should allow enabling or disabling of the RESTORE key, but any attempt to use RESTORE(FALSE) as a command or in a program was rejected. The RESTORE key is stated to be enabled, but I found in my version that it was disabled. The manual gives correctly an example of using the real-time clock of the C-64.

This, then, is as far as we can come on this exploration of Oxford Pascal, because like learning any other language, mastery now depends on practice. The aim of this book has been to guide your steps way from the problems that seem to plague other aspiring Pascal programmers. Many books tell you what can be done with Pascal; few mention what cannot be done and why. I hope

that by the time you read this, you will know well enough what you *cannot* do so as to be able to enjoy Pascal programming relatively free of error messages. That way, you get more practice and you learn more quickly. Happy programming.

Appendix B

STANDARD FORM

Standard form is a method of writing and working with numbers which are either very large or very small. The principle is that only a limited number of digits in a numerical quantity are really significant. For example, if a poll calls for 10 000 people to be questioned, it really does not matter very much if the number is 9 990 or 10 010. A number like 0.000 000 126 842 314 might just as well be written as 0.000 000 127 unless there is a special need to use more than three significant figures.

To see why standard form is used, consider the number 1 000 000. If you wanted to multiply this by 1.5, you would not (I hope!) go through the stages of saying: 'one point five times zero is zero, one point five times zero is zero ...' for six times before coming to 'one point five times one is one point five'. You would, of course, simply say 'one point five times a million is one point five million'. This is what standard form is about. Each number, when put into standard form consists of two parts. One part is a number which is greater than one but less than ten, and uses as many significant figures as is required. If you settled for four places after the decimal point, for example, you might use numbers like 1.237 5, 9.991 8, 6.417 5 and so on. The second part of the number is a 'power of ten'. This is really just a shorthand way of writing the number of zeros, and of indicating whether the number is greater than one or less than one. The number 10 is written as E1 (it has one zero), 100 as E2 (two zeros) and 100 000 as E5. Fractions are written using a negative sign, and a number which is *one more than the number of zeros*. Thus 0.01 is E−2, 0.0001 is E−4 and so on. A complete number in standard form would be written as 3.415 2E4 or 7.162 5E−3. These correspond to the numbers 34 152 and 0.007 162 5.

TO PUT A NUMBER INTO STANDARD FORM

1. Shift the decimal point until the number is between one and ten. Count the number of decimal places, and note if you have to shift left or right.

2. Write down the number as it is now, and chop off all digits beyond the limit that you have decided.
3. Write down the 'E' and, if you have had to shift the decimal point to the *right*, put a negative sign.
4. Follow this with the number of places that you shifted the point.

Examples

(a) 1 834 519 371 402.
The number of figures to be used is eight. Shift the point twelve places left, and write down eight figures: 1.834 519 3.
Now add E12 to give 1.834 519 3E12.

(b) 0.000 000 000 141 536 128 681 435.
Shift point ten places right, and chop to eight figures to get: 1.415 361 2.
then write E−10, so that the number becomes 1.415 361 2E−10.

CONVERTING BACK FROM STANDARD FORM

1. If the sign which follows E is negative, then the point is to be moved *left*, ELSE the point is to be moved right.
2. Move the decimal point, filling out with zeros when there are no digits to use.

Examples

(a) 3.124 567 4E8.
Move six places right to get 312 456 740. The zero has been added to make the eighth place.

(b) 4.198 302 7E−4.
Move four places to the *left*, padding with zeros, to get: 0.000 419 830 27.

INDEX

OXFORD COMPUTER SYSTEMS ORDER FORM

Please send me:

OXFORD PASCAL FOR THE BBC on:

[ ] . . . Cassette @ £39.95
[ ] . . . 80 track disc @ £49.95
[ ] . . . 40 track disc @ £49.95

(All the above include a 16K ROM)

OXFORD PASCAL FOR THE COMMODORE 64 on:

[ ] . . . Cassette @ £22.95
[ ] . . . Disc @ £49.95

All prices are inclusive of VAT. There is a charge of £2.00 for p+p.

I enclose a Cheque/Postal order/Access number

value £..... for the above goods.

Name *(please print)* .....

Address .....

.....

.....

.....

Telephone Number .....

Return to:
Oxford Computer Systems (Software) Ltd.
Hensington Road, Woodstock, Oxford OX7 1JR
Tel. (0993) 812700

OXFORD PASCAL ON THE COMMODORE 64

THE OFFICIAL GUIDE — recommended by Oxford Computer Systems (Software) Ltd*, the authors of Oxford Pascal

Explore the powerful Pascal programming language — and exploit the full potential of Oxford Pascal on your Commodore 64 — with the help of this readable guide. It provides simple explanations, clear program listings, plenty of practical examples, and useful advice on debugging techniques — all designed to help you make the most of Oxford Pascal for the Commodore 64.

Ian Sinclair is a popular authority on computing and has written a number of bestselling books. He worked for many years in industry and education before becoming a professional author.

Also by Ian Sinclair
Oxford Pascal on the BBC Micro

*Oxford Computer Systems (Software) Ltd, Hensington Road, Woodstock, Oxford OX7 1JR, England. Telephone (0993) 812700.

ISBN 0-304-31267-3

9 780304 312672