advanced programming techniques

on the commodore 64

powerful ideas and applications

david lawrence



CASHTRONIC DATA
Vestergade 35
4850 STUBBEKØBING
Tlf. 03-84 10 95

advanced programming techniques

on the commodore 64

powerful ideas and applications

david lawrence

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

Reprinted 1984

Copyright © David Lawrence

ISBN 0 946408 23 8

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

Cover Design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by The Leagrave Press Ltd.

CONTENTS

	Page
Notes on Program Listings	ix
Introduction	1
1 Modular Programming	3
2 Debugging	17
3 Strings	27
4 Inputting Information	41
5 Error Trapping	53
6 Storing and Retrieving	65
7 Logical Conditions	79
8 A Jungle of Sorts	93
9 Data Structures: I	111
10 Data Structures: II	127
11 Inserting Data	141
12 Odds and Ends	149
13 Formatting	159
Postword	171

Contents in detail

CHAPTER 1

Modular Programming

A guide to writing programs that work — defining your program — planning the program — writing the modules — entering the program — hints and tips.

CHAPTER 2

Debugging

Information — single line error messages — non-obvious error messages — interpreting error messages.

CHAPTER 3

Strings

Concatenation or string addition — string subtraction — inserting items into a string — moving items within a string — searching within strings — regular string structures — multiple element string arrays — data in variable length strings — garbage collection.

CHAPTER 4

Inputting Information

Entering information: INPUT — simple entries with INPUT — INPUT of several items using the same screen line — INPUT to screen boxes — GET — GET and creating a waiting state — moving a cursor with GET — GET and timed responses — GET and inverse boxes — screen editing with GET — simple screen editing using INPUT.

CHAPTER 5

Error Trapping

Avoiding errors — error trapping — setting limits — garbled entries — common sense error-trapping — DIY error messages.

CHAPTER 6

Storing and Retrieving

Saving programs — saving and loading data — saving to tape — printing to a file — loading from tape — Save and Load routines — variations for disk.

CHAPTER 7

Logical Conditions

The humble IF — protecting against illegal values — errors arising from IF — IF...THEN...ELSE — IF with >,< and = — IF with the operators AND and OR — combining AND and OR — unpacking complex conditions — setting limits — IF with NOT — using logical conditions — the value of a condition — using conditions as values — plus or minus? — multiplying and dividing — avoiding isolation by IF — AND and OR with numbers — POKE with AND and OR — storing with AND/OR — odd or even?

CHAPTER 8

A Jungle of Sorts

The whys and wherefores of sorting — the Bubble Sort — programming the Bubble Sort — the Delayed Replacement Sort — the Shell-Metzner Sort.

CHAPTER 9

Data Structures: I

Simple data structures — data structures for numbers — single byte numbers in integer arrays — storing directly in free memory — numbers in strings — stacks — string data structures — packed strings — packed strings using numeric array pointers.

CHAPTER 10

Data Structures: II

Linked lists — pointer strings — deleting with pointers — the black hole problem.

CHAPTER 11

Inserting Data

Normal search and shift — binary searching — pure searching — binary searching with pointer arrays.

CHAPTER 12
Odds and Ends
User defined functions — terminating FOR loops — DATA statements — timing with TI and TI$ — rounding with INT.

CHAPTER 13
Formatting
Cursor controls — use of cursor control characters — TAB — SPC — simulating PRINT USING — justifying — easy logos and designs.

Notes on program listings

For the sake of clarity, control characters in the program lines which follow have been set out as follows:

CURSOR UP	[CU]
CURSOR DOWN	[CD]
CURSOR LEFT	[CL]
CURSOR RIGHT	[CR]
CLEAR SCREEN	[CLR]
HOME CURSOR	[HOME]
REVERSE ON	[RVS ON]
REVERSE OFF	[RVS OFF]

Colours are represented by the colour name in square brackets. Thus the control character for yellow is:

```
[YELLOW]
```

Where a number of control characters are required the format is, for instance:

```
[18*CD]
```

Within program listings, single quotes are used in place of double quotes and should be replaced when listings are entered.

Introduction

This book is intended to be unlike any other you have on your shelf. It is not a collection of programs, it is not an introduction to the commands available in BASIC. It is not a collection of trivial routines such as how to write a two line subroutine to convert Fahrenheit to Centigrade.

The book is dedicated, like its predecessor *The Working Commodore 64,* to those who want to set their micro loose on useful tasks, the kind of uses that are normally termed 'applications programs'. The success of *The Working Commodore 64,* already into its fourth edition, shows that the era when people were happy to use the power of a micro-computer only for games is over for good. People *want* to set micros to work: the problem is how to design the programs that will do it.

There is a lot of mystique attached to applications programming and most of it is bunkum. An applications program is, at its simplest, something that allows the input of information, stores it, processes it and then outputs it again in some useful way. The information may be names and addresses, products and prices, financial records, the list is endless. How the information needs to be processed is also infinitely varied, some will simply have to be stored, some sorted into a desired order, some will have complex mathematical procedures performed on it. No book can claim to give guidance on all the different ways to process the information that a micro can store. It depends on the information and the purpose behind storing it.

What can be done is to give some guidelines for the processes that will go on alongside the core of the program. How to design a program so that it is more likely to work, how to debug it, how to accept information and deal with errors, economical and fast ways to put information into the memory, how to sort, how to format output so that it is clear and comprehensible. The lion's share of the programming for any serious application will be made up of such things and this book sets out to explain how they can be achieved economically and successfully.

The complexity of the material in this book varies immensely. In the chapters that follow you will find many techniques which require only a single line of program, two or three line routines to achieve simple objectives like aligning the decimal points on a series of numbers, lengthy and complex routines to allow data to be inserted into large arrays at high

speed. They are all included because they are useful, for this is not intended to be a book of theory. What I have tried to do is to look at the techniques I use myself and at the work of other people and to find common themes and methods. There is nothing in the book that I have not seen used to good advantage in solving a particular problem.

The book does not attempt to deal with the use of mathematics in micro-computing. That is a specialist area worthy of at least a book in its own right. Of course there is a little bit of mathematics from time to time but only as much as is absolutely necessary to make something work. If you want to go further into mathematics I suggest you buy Czes Kosniowski's new book for the 64, *Mathematics on the Commodore 64.* Apart from what is necessary to obtain a clear and attractive display of information the book does not attempt to deal with graphics techniques, again a specialist area of its own.

My thanks go to all the readers who have encouraged me to write a book like this one: I hope it is at least partly what they were hoping for. Thanks are also due to Mark England, co-author of *Commodore 64 Machine Code Master,* for waiting so patiently for this to be finished before starting on our next book together. Finally, and most importantly, thanks to Barny and Tom for understanding that sitting in front of a computer can sometimes be more important than playing and to my wife Jane for being a rock on which everything else can be built.

I hope that the final product is worth all the encouragement and help. I hope that it is a book you will come back to time and time again to shed some light on the inevitable problems that programming brings. Most of all I hope it is a book that will give you ideas. It is a book of tools for programming and it will have done its job not when you understand those tools but when you put them to new uses.

CHAPTER 1
Modular Programming

A guide to writing programs that work

It may seem strange, in a book about techniques of BASIC programming, to begin with a chapter which is not so much about BASIC but about the problems of program style and layout. There is, however, a simple reason why this chapter is included. As I deal with questions and problems from micro-owners who are writing their own programs, I become more and more aware that often the only thing holding them back is not the lack of understanding of how the BASIC language works but a confused approach to the actual task of putting BASIC to work in a program.

The technique of programming that I always use myself, both in books and in my own programs is called 'modular programming'. At its simplest that means writing programs which are made up of self-contained sections. Most of the practical examples given in this book are written in that form. They can be lifted from the page and grafted easily into whichever of your own programs that might need them. Modular programming, however, goes much further than that. It implies a whole philosophy of programming style and if that sounds too grand then what we are really talking about is applying common sense to the writing of computer programs.

In this chapter we shall discuss some of the steps involved in writing a successful program, steps which begin, or at least should begin, long before you ever touch the keyboard of your 64.

Defining your program

The worst programs that anyone ever writes are the ones that they are enthusiastic about before they start. They get an idea into their head and, full of confidence, they rush to the keyboard and try to put the vision into practice. If they know what they are doing then they quickly manage to put the core of the program into working order. Then they realise, for instance, that there is no way of properly inputting data so they find a space in the program or tag onto the end a routine to do that. Then something is added to take account of the need to remove incorrect items, then something to deal with invalid inputs which would crash the program (which always happens when someone else gets their hands on it). Then of course there have to be a few lines added to store some data on tape.

Then there's the problem that no-one else understands how to use the program so a better 'menu' has to be added and perhaps some instructions. Then there is...

By the end of the process the program consists of a dog's dinner of disorderly line numbers and tangled GOTOs pointing here there and everywhere. When that inevitable bug is encountered, it becomes not so much a problem of identifying *what* it is as one of discovering *where* it is, a process that can take days or even weeks.

The best programs are the ones that people know they have to write but don't really want to or don't really know how to. The reason is simple. When you are not confident that you can write the program you want then you sit down and think about the task. The moment you do that you have crossed the first and in many ways the most important hurdle in designing a successful program.

A program does not simply consist of the method of accomplishing its central task, like calculating tax or storing data files of names and addresses. It is a complex of functions which must cope with the input of information and the proper display of that information, it must be capable of making clear the way in which it is to be used, it must deal with errors, it must allow corrections to be made if incorrect information is input. All of these tasks, and many more, are just as central to a useful program as the routine at its core.

Even when you consider only the central task and the lines necessary to accomplish it, the first idea that you have is seldom the best. So you want to calculate your tax, but what do you want to include in that ability. Do you want to be able to look at the effect of different tax rates. Do you want to allow for expenses to be deducted. How long a period do you want to cover and, if it is more than a year, what happens if tax rates vary from year to year (or even during a year). Do you want the program to be able to handle 'what if' questions like 'what would happen if my income went up by 78.70 per month', without affecting the data you have already input. Do you want to be able to store the information or will you put it in afresh every time you run the program. Is your spouse's income to be included. All these are questions which I include 'off the top of my head' as it were. A couple of hours' thought and you could fill a page with the things you need to know before you can sit down and begin to program something that seemed perfectly obvious when it first occurred to you. Failure to actually think the purposes of the program through may well still allow you to write something that works but discarded cassettes all over the world are full of programs that work but are not really useful.

The first task, then, in writing a program is to sit down and think. To write down in plain English what it is that you want the program to be able to do. Once that is done, leave it for a while and come back to it. It will seldom seem as perfect on the second examination, so a few

functions will be added. If you are writing a program which is going to be used by someone else, however infrequently, then you must also take into account what they will expect it to do. It is no use whatsoever trying to convince your children that the brilliant new educational program you have written is exciting and enjoyable, if it doesn't do what they would really like it to. A program either does what people want and expect or it doesn't. If it doesn't then you have wasted a great deal of time in writing it.

The conclusion is that you should always overdo your program specification, plan it with far more than the basic necessities in mind. It may be that when you come to the detailed planning and writing, some things will have to be dropped for lack of memory or lack of knowledge. More often, you will find that you *can* write the program you really wanted. It is the programs which cover all the ground which are the ones you will come back to time and time again.

Planning the program

Planning? Hasn't that already been covered? Well no, because up to now you haven't even been thinking about a computer program. What you have been defining is an ideal slave. Something that will perform a task exactly as you want it performed. You have paid no heed, if you have done the task properly, to what you know how to do, or indeed what can be done on the 64. The task now is to break up your ideal specification into units which the 64 can handle and which you can program.

This is probably best done as a two-stage process. The first stage is defining the broad areas of the program, like input, output, data processing, storage and so on. Once that fairly simple task is achieved comes the more complex one of identifying the actual units, or modules, out of which the program will be built up.

The rule here is to split the functions of the program as far as possible into separate units, even if that means splitting functions which seem always to go together. A program which is strictly divided into functional units is always the easiest to write and always the easiest when it comes to the inevitable process of debugging. In addition, paradoxically, programs which are properly divided into functional units will often be shorter than those where all the program functions are tightly packed together. The reason for this is that if you break down the program properly you will find that certain functions are used time and time again by different parts of the program in different places.

Take the example of our hypothetical tax program. For the screen display of the data you will need to write a little routine to format the figures to two decimal places and in such a way that all the decimal places are aligned when the figures are printed in a column (see Chapter 13).

You could tag the three or four line routines onto the output part of the program with no problems. However, then you find that when deleting items you would like to display them in the same format so that the user can see exactly what is being deleted. The same goes when something is input — you would like to echo it to the screen in formatted form. Because you have embedded your little formatting routine in another part of the program, there is no way that you can actually use it from anywhere else and you have to write it again each time you need it.

That may sound ludicrous but look at the kind of programs published in the back of computing magazines and you will find that kind of repetition happening all the time. In a properly written program every function can be accessed from anywhere in the program. Indeed, very often you will find that in a properly written program new capabilities can be added simply by calling a series of functions in a new order.

Most important of all in the long term, once you have three or four programs where every separate function is clearly identified, then the task of writing subsequent programs is enormously simplified. There are really very few important techniques in serious programming and once you have written them into a program you will find yourself lifting them out and using them to assemble further programs with the same ease that you would use a child's construction toy.

The basic technique for achieving a properly structured program is to describe each aspect of the program to yourself:
'In this section of the program I want to give instructions about the way data should be input, accept data from the user, check the data for certain errors, ask the user to confirm that this is the data intended and then put the data into the correct place in the memory.'

In this simple, relatively non-technical sentence you have already identified at least five functions and you should immediately write them down. Go through all of the broad program areas in this way and you will end up with a list of functions which look sufficient to provide a program.

Now comes the task of identifying the functions which could be broken down further. For instance, we mentioned earlier that the input would be echoed to the screen, presumably for the user to confirm that the input was correct. If you want to format that display then there has to be a format routine, as discussed above. Another example would be the last function mentioned, inserting the data into the correct place in memory. If you were ordering items in alphabetical or date order, then inserting a new item would involve first searching for the correct place and then making room for the new item. These are two different processes, as you would find out when later you want to allow the user to specify a particular item which will be recalled and displayed, only to find that your search routine is tied up with lines that also insert an item.

As with the overall program areas, go through each of your program

functions and describe them to yourself. If you end up talking about two or three distinct actions for a particular function then it is possible that you have identified a candidate for further division.

At the end of this process you should end up with a list of functions which you think your 64 will need in order to execute the program you want. You will have a picture of the whole program, the kind of length it will be, the sort of structure it might have, the areas you might have problems with. As a final check, run through a few typical operations you would like to be able to perform with your program, using your notes as a substitute for the 64:

'Switch on, RUN (whoops no menu to tell me what it does), specify that I want to enter data, confirm entry, enter another item.....'

If all is well, your notes should function almost as well as your 64 eventually will.

Writing the modules

By now, the temptation to reach for the 64 and start banging away at the keyboard is almost irresistible, but the time is still not right. The worst place to write a computer program is on a computer, the best place is on a piece of paper. By this I don't mean that every line of the BASIC which will make up the program must be noted down perfectly before anything can be entered. This is the stage at which you need to design a working model of each of the separate program functions. These models will guide you when you come to enter each function as a module of the program but they will probably not be anywhere near as detailed as the module itself.

Probably the best-known method of developing a program in some detail is the use of flowcharts. The problem with such charts is that no matter how much computer professionals advocate their use, micro owners seldom if ever use them. More realistic, I think, is to use what is known as a 'program development language' or PDL. This may sound daunting, and indeed in the hands of many high power programmers PDL has become a technical tool which is very difficult for most BASIC programmers to understand, let alone use. I am not referring to such a complex beast but to a straightforward mix of BASIC and English which will look something like the finished module but is much quicker to write.

In writing an input module I might end up with something like this:

```
## PRINT [TITLE]

   PRINT [COMMANDS AVAILABLE] NEW ITEM/QUIT
```

```
%% INPUT 'AMOUNT'; TA
   IF TA = -9999 THEN RETURN
   GOSUB ERROR CHECK
   IF ERROR GOTO %%
   INPUT 'DESCRIPTION'; TD$
   GOSUB FORMAT
   CLEAR FROM MENU DOWN
   PRINT TD$;':';FRMAT$
   INPUT 'CORRECT'; Q$ : 'N' GOTO %%
   GOSUB PLACE SEARCH
   GOSUB DATA INSERT
   GOTO ##
   [VARIABLES REQUIRED: none]
```

VARIABLES USED:
TA=temporary storage for amount
TD$=temporary storage for description
Q$=temporary input
FRMAT$=formatted amount provided by format routine

Written in this kind of form I can see exactly what the module is intended to do and I have a perfectly clear idea how I am going to program it. Notice that I have spelled some things out in full because it is just as short as describing them in English. Other functions, especially those referring to the display of items on the screen, it is easier just to mention and leave it to the final entry of the module to decide how many lines of display have to be cleared if the item has to be re-input or what colour the title and menu are to be printed in. Sometimes I may well include a one line description of a procedure that I can't see at the moment how to write but know is soluble with a little thought and experiment. In these cases a note can be made next to the line and a supplementary sheet can be

written later.

There are no line numbers, partially because I haven't attempted to spell out what every line will be but also because it would slow down the writing. One or two labels at the beginning of lines such as ## and %% are enough. When I come to actually enter the module on the 64 I shall put GOTO ***** for jumps forward and then re-edit the line when I know the number of the destination line. Jumps back will be no problem since the line number of the destination will already be known. You will also note that the destinations of the GOSUBS are not specified, merely the names of other routines that I may or may not have written yet. When it comes to entering the program I shall first decide upon a block of lines for each module, preferably a separate block of at least 1000 for each, and will note the location at the top of each of the separate sheets containing the program functions.

At the end of the listing you will notice that there is a list of the variables required. Some of these will be used to remind me of the variables which must be declared before the module can be called up. They will also be used to make up a list of variables so that I can determine that I am not duplicating names. Temporary variables, which are used only for the duration of the module and whose contents are either forgotten or transferred to another variable for permanent storage, can have their names duplicated in other modules. Others will cause chaos if they are inadvertently used for different purposes at different times.

Clearly, such a happy-go-lucky approach will not work for every module. Some of the modules may contain one or two lines of mathematical calculation and these I would spell out in full. In general however, a method like this, freely adapted to your own needs, will enable you to build up a clear picture of the program and then to enter it far more quickly than if you went directly to the keyboard. It is not intended that you should slavishly copy the style shown in the example. All that is necessary is to recognise that a program *can* be written quickly and understandably by adopting this kind of approach.

Entering the program

Now you can turn to the 64 in the confidence that you have something to enter which is close to a working program. Your raw materials will be a sheaf of notes containing the various functions of the program. What you do *not* want to do, however, is to start at the beginning of the program and enter it straight through to the end.

Modular programming lends itself almost perfectly to the debugging of programs as they are entered and it is far easier to debug a program as you enter each module than to track down errors when the whole

program is running. Your debugging will never be perfect at this stage since some errors will only become apparent when all of the modules are interacting with each other but you can save yourself a lot of heartache nevertheless. In order to debug a program as you go, you need to identify which are the modules which are most often used, probably for trivial purposes, and enter them first.

In the case of the input module listed above, it could be tested without the presence of the SEARCH PLACE or INSERT DATA routines. They could simply be replaced by two RETURN statements at the appropriate start lines of the blocks they will eventually occupy. You could not, however, test the input module without having first entered the ERROR CHECK and FORMAT routines. Having entered these two you will be able to input data to your heart's content. Nothing will be done with it but you will be able to see that the screen display is clear and that inputs are accepted and tested.

It is never possible to get the sequence in which you enter the modules exactly right. You will often find yourself having to declare the value of one or two variables in direct mode (eg LET A$='TAX REBATE' without a line number) and then entering GOTO the start of the routine (RUN would wipe out the variable you have just entered). Sometimes you will have to enter two or more modules together, such as SEARCH PLACE and DATA INSERT, since they will normally work in tandem. Despite these exceptions to the rule you should always attempt to test everything as soon as possible after you have entered it, knowing that every error you discover is going to be relatively simple to track down and correct since there is a 95% chance that it is in the last module(s) you entered.

The result of all this, as I have said, is not going to be a bug-free program, but there will be far less bugs than if you were to bash away from start to finish. At the very least you should end up with a complete program without a single SYNTAX ERROR, since every line of the program will have been run before the program is completely entered.

Hints and tips

Given below are a few of the things to watch out for when writing a program in modular form. The list is not exhaustive but represents the kind of points which are all too often neglected in micro programming:

1) Programs are clearer if all modules are given an explanatory heading. I always use a format such as the following:

```
1000 REM*********************************
1001 REM NAME OF MODULE
```

```
1002 REM*********************************
```

This clearly marks out the module in the program and will be invaluable when you come back to the listing after some time.

2) Having put a heading on a module, always refer to the line number of the heading in any GOTOs or GOSUBs to the routine. The reason for this is that you may well want to add a new first line to the working part of the module, or delete the present first line. This will leave you with UNDEFINED LINE errors cropping up every time the module is called. The position of the heading will never change, so any modifications you make to the main body of the routine will cause no problems.

3) Do decorate your program liberally with comments contained in REM statements whenever you do anything in the least obscure. Three months later, when you want to modify the program, you will bless the day when you took the little extra trouble involved. Without comments it could well take several hours to work out again just what is going on.

4) In any program, especially a substantial one, descriptive variable names make a program easier to understand. Such names can, however, produce some interesting errors if they contain combinations of letters which are also the start of BASIC keywords. Note that in the listing given above, FRMAT$ has the 'O' missing so that it is not rejected because of the 'FOR' at the beginning. The other problem with descriptive names is that it is very easy to duplicate the effective part of the name (the first two letters) without realising it eg PAYMENT and PARAMETER are actually the same variable. This is where the list of variables made up during the writing stage should prove invaluable.

5) The majority of variables used in a modular program will (or should) be temporary variables. If you accept an input straight into the array or variable which contains your permanent data it is much harder to correct if it turns out that there is an error in the input. Inputs should be made to a temporary variable and there is no need to call these by different names in each module, normally one or two such as T$, Q$, T, and Q will do the job without confusion since they will be forgotten at the end of the module.

6) Some variables are less temporary than others. In the input module listed above the data was stored in two variables (TA and TD$) which were actually going to be passed to two other modules before being placed into the main data arrays of the program. In this case it is helpful to begin the variable name with T to show that this is a temporary variable

but do add one or more letters which will remind you of its function, always remembering the danger of duplication of names.

7) Take some care in deciding the order in which modules are laid out in the program. There are two major considerations here:

a) Frequently used modules will be executed slightly faster if they are placed towards the beginning of the program.

b) On the other hand, there is nothing more difficult to understand than a program where all the modules are apparently jumbled together in no logical order.

If you are not doing anything excessively complicated, with enormous amounts of calculation and time then it is probably better to lay out the program so that the modules fall into logical groups, with smaller modules which are used by several areas of the program (like the FORMAT routine in the example given above), at the end. It is extremely unlikely that you will notice any difference in speed.

8) The initialisation of the program, ie the declaring of the various arrays and variables, is a separate function of the program and should have a module of its own which can be called or not. This will allow you to set up the program when it is first used and then to wipe out any data if it is subsequently called. Programs can also be made to 'auto-initialise', ie to set the arrays when they are needed but to avoid clearing the memory if there is some useful data already stored. In order to do this, place the section of the program which clears the memory and dimensions the arrays at the very beginning of the program. At the start of the module, have a line which tests an important variable to see whether it is zero. The variable chosen should be one which, when the program has data in memory, is always some value other than zero. What the auto-initialise line does is to jump around the initialisation routine if the variable is *not* equal to zero. Take the following example:

```
1000 REM*********************************
1001 REM INITIALISE
1002 REM*********************************
1010 IF IT<>0 THEN 1500
1020 CLR
1030 DIM A$(10), A%(500),B%(100),C$(100)
```

Here, IT will be used to record the number of items stored by a program. If you were to input some items,stop the program and start it again with

GOTO 1000, instead of the data being lost, line 1010 would jump around the lines which clear the memory and set up the arrays afresh. When you want to clear the existing data, all that needs to be done is to start the program with RUN, since this clears the memory and sets all variables, including IT, to zero.

9) Any program worth the name needs a menu which specifies, at least in outline, what the program does and allows you to choose between the various functions. Apart from that main menu, many of the individual modules could probably be improved by means of a small menu if they allow more than one function to be accessed.

10) Modular programming is made easier by the use of 'flags'. These are variables used to indicate to one module that something has or has not happened in another. The classic example is that of error flagging. Modular programming lends itself to the use of a separate routine in the program to declare different types of errors, but how is that routine to be called, and how are you to prevent an error crashing the program. The answer is the use of one or more flags.

For example, suppose that you are four subroutines along a chain of GOSUBs, ie the first routine has called the second, the second has called the third and the third has called the fourth. At that point an error is found in the data which could not have been detected before, say you are about to insert an item and it is found that there is no more room in the array. You now want to do two things: firstly to notify the user of the problem and secondly to ensure that nothing disastrous happens when you RETURN from the current subroutine. This is usually done by means of an error flag, say a variable called ERR. This would normally be set to zero but is now given a value corresponding to the type of error. All the modules will contain one or more lines which detect whether ERR is still zero. If it is not they RETURN execution to the previous module in the chain, without trying to do anything. At the beginning of the chain will be a module which will detect that ERR has been set and will call up the ERROR MESSAGE module to print out error message number ERR. This is only one example of the use of a flag, you will find them invaluable whenever information on the state of play has to be passed form one module to another. (For a fuller explanation of the use of an error flag see Chapter 5).

11) Spread the start line numbers of your modules. When you want to develop the program further, and you no doubt will, there is nothing more annoying than having to spoil your neat structure by squeezing a new module in with line numbers incremented by 1 each time or adding new modules on the end which would more naturally be part of a group

in the middle of the program. A spread of 2000 for each module is not too much to begin with for many programs.

12) There are circumstances when the execution of a program can be simplified by replacing RETURN statements with GOTOs. For instance Module 1 calls Module 2. On a certain condition being detected, instead of RETURNing to Module 1, we need to call Module 3. One way would be to call Module 3 with a GOSUB, then RETURN to Module 2, then RETURN to Module 1. This can also be accomplished by a simple GOTO calling Module 3, where the RETURN statement at the end of the module will send execution back to Module 1 without having first to RETURN to Module 2. An example of the use of such a technique would be with the error-checking routine described above. If we regard any module which is called directly from the main menu of the program a 'second level' module, then every second level module could have an error-detecting line which would GOTO the error message module and then RETURN to the menu. The technique has to be used with caution, however, since miscounting the number of GOSUBs and RETURNs involved can either lead to the program stopping with an OUT OF MEMORY error when it runs out of space to store GOSUBs which have not been cancelled by RETURNs, or executing a subroutine you had not expected.

13) Once you have written a program in modular form, remember that it is easy to change. Be on the lookout for better techniques of performing a certain function. When you find an improvement in a book or magazine, whip out the original module and replace it with one embodying the new technique. Failure to update your programs means that you are missing out on one of the greatest advantages of this style of programming.

14) For modules embodying important techniques it is often advantageous to store them separately as well as included in the program. This will enable you to add them easily to subsequent programs, either by using a 'merge' program like that contained in *The Working Commodore 64* (Sunshine Books: 1983) or simply by loading the routine from tape or disc, listing it to the screen, loading the new program and then running the cursor down the lines on the screen using the RETURN key. This will, of course, insert the lines on the screen into the new program. Remember that you may need to change the line numbers on the module before it is inserted.

15) Finally, remember that almost everything can be modularised. Good professional programs will often contain a great many separate subroutines of only two or three lines. It is possible to go too far with the

process, but not easy.

Conclusion

There is little doubt that this is the easiest chapter in the book to miss out. It contains far less by way or practical examples and no working BASIC. Only by putting the principles of this chapter into practice will you learn that, of all the material contained in this book, it is this chapter that will make the biggest difference in your attempts to write successful, useful and understandable programs.

CHAPTER 2
Debugging

A national computing magazine recently ran a series of articles explaining the many different computer languages available today. When the series had ended a reader wrote in to point out that they had missed the one language familiar to all computer owners whatever their machine —profanity.

When you have entered your program, testing it as you go, when you have even run it once or twice without apparent problems, there will nevertheless come a time when it breaks down in chaos. This is bad enough on a program of your own, whose working methods you understand. It is even worse if you are entering someone else's program, perhaps from a book or magazine. With the best will in the world you will not have developed an instinctive feeling for what is going on where. I know that some readers of my own books have spent weeks or months trying to debug a program they have made an error in entering. By the time they contact me they are often near to desperation, convinced that their 64 is malfunctioning (or that I am). The truth is that in most cases they could have solved their own problems in a few minutes if only they had known and used a few simple techniques.

Information — the key to debugging

The first thing to remember when you encounter an error in your program is that all the information you require to track down the error is contained safely in your 64 and the last thing you wish to do is to erase those priceless facts. When you find an error, therefore, never ignore it and run the program again in the hope that it will not crop up next time. You may well be right — the program will apparently run perfectly but you have lost the chance to remove a bug which will inevitably rear its head again in the future, perhaps at some time when you have valuable information contained in the memory. The first rule of debugging is, then, to trace each error as you find it.

Having made that resolution, how are you to go about making the best use of the information available? To decide that, you need to assess what exactly that information is:

1) It may well be that the type of error that has stopped the program is such that it pinpoints the problem as being limited to that particular line. In that case the 64 has identified the type of mistake you have made and the location in the program at which it occurs. The error report will take the form:

```
? [ERROR MESSAGE] IN LINE XX
```

2) Some errors may stop the program even though the line on which it stops seems perfectly correct. Along with these go errors which will not stop the program but simply result in the program producing a nonsense result. This type of error is the most difficult to trace, since there will often be no indication of where things are going wrong.

Single line error messages

In most cases, when you get an error message, half your problems are solved because the 64 has already pinned down for you the program line that needs to be examined. It may be that the actual correction to the program has to be made to some other line(s) in the program, but the key to discovering the nature of the error is always contained in the line indicated in the error message. For many of the error messages you will need to look no further than the line mentioned, a fact that people often refuse to recognise.

Take, for instance, the SYNTAX ERROR message, the commonest of all errors. Whenever you receive this error you know that there *must* be an error in the way you have entered the line specified. It is no use perusing the line in question, deciding that it looks OK and then running the program again or trying to alter other lines to make the program work. There is something in the line specified which the 64 does not recognise as BASIC and the program will never run until that line is corrected.

Error messages which normally pinpoint the exact location of the error are: SYNTAX ERROR, FILE NOT FOUND, FORMULA TOO COMPLEX, REDIMMED ARRAY, TYPE MISMATCH, UNDEFINED FUNCTION, UNDEFINED STATEMENT. The procedure with such 'line specific' errors is as follows.

1) List out the program area which precedes the line, this will help to put the line in context.

2) List out the line itself again, this time on its own, with one or two blank lines separating it from the lines you have already listed. The reason for this is that it is surprisingly easy to enter two lines on the 64 in such a way

that they are received as a single line. Such errors are almost impossible to detect unless a line is listed separately.
Examine the following:

```
10 FOR I=1 TO 10 : LET X=I*100-50/(I*2)
20 NEXT I
```

Imagine the frustration when it is run and all that ever happens is that ?SYNTAX ERROR IN LINE 10 flashes up on the screen. The search for the error can go on for hours or days. In fact all that has happened is that at the end of line 10, instead of pressing RETURN, a space was entered, thus taking the cursor to the beginning of the next line, then line 20 was entered. To the 64, the full version of line 10 reads.

```
10 FOR I=1 TO 10 : LET X=I*100-50/(I*2) 20
NEXT I
```

Not surprisingly, the 64 finds that a little hard to interpret.

3) With the line listed out and provided that there are no gross errors, the task is to examine the line instruction by instruction and character by character. One way to ensure that you don't skip too quickly along the line is to place the cursor at the beginning of the line and then, using the 'cursor right' key, to move slowly along the line, examining each character and the instruction of which it is a part. Most syntax errors (and the other errors which result from the 64 not understanding the line in the way you intended it) result from the dropping of characters (especially brackets), the misspelling of keywords or the inadvertent transposition of characters. Particular points to watch out for are missing colons between instructions, the figure 1 replaced by the letter 'I' and zero replaced by the letter 'O'. For instance:

`GOTO I000` instead of `GOTO 1000` would produce
`UNDEFINED LINE ERROR.`

4) The detailed scan of the line will often fail to turn up the mistake. In that case the next step depends upon whether you need to preserve the value of the variables in memory or not. In the case of a syntax error the value of the variables will be irrelevant and you can afford to make changes in the line to help track down the error, even though this will clear the variables area. If the value of one or more variables does seem to be involved in the problem, move on to the section headed 'Non-obvious error messages' later in the chapter.

5) Provided that the variables are not needed, place a STOP statement in a new line just after the line where the error was indicated. Now start at the last statement in the offending line and insert a REM at the very beginning of the statement. Run the line again (in some cases you will have to run the program again up to that point). If the syntax error has disappeared, then the error is in the last statement, since the REM has effectively removed that statement from the line. If there is still a syntax error then delete the REM and re-insert it at the beginning of the previous statement in the line. By the time you have reached the first statement in the line you will have identified the statement which contains a syntax error.

6) If you still cannot identify the error, then begin to change the names of variables and try again. Remember that for some errors the actual value of the variables does not matter in the least. It is the *form* of the line that is wrong. The purpose behind changing variable names is to try and see whether you have entered invalid names, perhaps names which begin with the same letters as the start of a BASIC keyword.

Non-obvious error messages

As mentioned above, some error messages do not pinpoint the exact location of the error in the program, they simply tell you where the key to the error is to be found. The distinction is not an absolute one. If you enter a line which reads:

```
10 A=10/0
```

then you will receive a DIVISION BY ZERO error message, and the reason will be easy to find. On the whole, however, BAD DATA, BAD SUBSCRIPT, DIVISION BY ZERO, FILE NOT OPEN, FILE OPEN, ILLEGAL QUANTITY, NEXT WITHOUT FOR, NOT INPUT FILE, NOT OUTPUT FILE, OUT OF DATA, OUT OF MEMORY, OVERFLOW, UNDEFINED FUNCTION and STRING TOO LONG, will most often occur on occasions when the real error is not in the program line indicated, but before it in the execution of the program.

The procedure in such cases is less clear than in debugging a single line. In the majority of cases the first step will be to print out the value of every variable contained within the line. Thus if the line read:

```
100 A=X*Y/(T1*T2)
```

you would enter

```
?X

?Y

?T1

?T2
```

and make a note of the values that this produced. Now work through the line in your head or on paper to see why it is that those particular values produced the error report that stopped the program. Until you can identify the reason you can go no further in tracking down the error. Normally this will not be a problem but if, due to the complexity of the line, you are unable to see how the variables work together, eventually you may have to resort to re-entering the instructions in the line but this time distributed among two or three separate shorter lines, then run the program again. This should only be done as a last resort, however, since unless you can reproduce the exact chain of events which led up to the error the error may not be repeated and the program bug will lie in wait for some future occasion.

Having identified the variable which is at fault the only course is to mentally trace back the execution of the program to see where the variable may have picked up the offending value. Unfortunately this is often not possible in a complex program. In this case the solution is to alter the program so that regular checks can be made of the value of the variable as the program is running. This is achieved by inserting new lines in the program after those sections of the program where the variable might be altered, each new line reading simply STOP — the drawback here is that inserting the lines will clear the memory and thus you will have to start the program from scratch.

Having entered the temporary stop lines the program is rerun in an attempt to repeat the error. Each time the program STOPs, you will be able to print out the value of the offending variable and then enter CONT to continue the program if you have not yet found the fault. If, when you find the area where the problem appears to be generated, the program lines seem correct, one factor to watch out for is the duplication of variable names. The lines referring to an important variable may in fact be faultless but the program rendered nonsense by the use of the same variable name for another purpose at some stage — remember that in all variables it is only the first two letters which are significant.

In all of this the task can be made much more simple if two rules are observed:

1) Where complex sets of data are going to be handled, one of the first program modules you should enter is the data file routine to store the information on tape or disc. When you enter data to begin testing, save it regularly so that, if the program does stop, you will be able to reload the last set of data from tape instead of entering from scratch from the keyboard.

2) Most program faults will show up just as well with a little data as with a large quantity. Rather than bash away entering huge quantities, enter only three or four items and then go through all of the program functions. If there is an error then it will be easy to simulate the sequence again if only four items have been entered, especially if you enter stylised items or values like AAAA, BBBB, CCCC, 1111, 2222 and 3333.

Searching for an error embedded in the program at some unspecified point can be a taxing task. It can only be carried out successfully if it is done thoroughly, following through the program's execution in detail, and only if you understand exactly what each program section is intended to achieve. It is when bugs like this crop up that you will be particularly thankful if you have followed the advice given in Chapter 1 and written your program in strictly functional modules, for such a program structure makes the tracking down of errors considerably easier.

Interpreting error messages

Program bugs are as varied as programmers, for the simple reason that they are entirely the work of programmers. For that reason it would be impossible to give an exhaustive list of what every particular error message might signify. Given below you will find a list of the common error messages and some suggestions as to likely causes:

BAD DATA : The usual cause is a failure to mirror exactly the structure of the save routine in its load counterpart. Another cause may be that the data saved has not been properly separated by CHR$(13) after each item. Be careful in using a variable to separate the items (eg PRINT#1,A$,R$,B$,R$,A,R$,B,) that you have actually defined the separator. The module may look fine but if R$ has not been defined as CHR$(13) then data is being run together on the tape or disc and the load module will end up out of sequence with what was saved.

BAD SUBSCRIPT : Look to the values in brackets after the array name, because one of them is larger than the respective dimension you gave to the array when it was DIMensioned. If the fault is not immediately obvious, check that you did dimension the array, since the program will accept references to elements zero to nine of a single dimension array

even if it has not been DIMensioned, but will then stop in confusion when you try to refer to element 10.

DEVICE NOT PRESENT : Either you have specified the wrong number in a statement which deals with a file (eg OPEN 1,7,1) or you have forgotten to plug in the disc or cassette recorder the program is trying to access. Unfortunately this error can also crop up if other file handling errors occur and the Input/Output system becomes confused. In some circumstances this can be cured by switching off the device in question, in others the 64 must be switched off and on again, with the consequent loss of the program and data.

DIVISION BY ZERO : It is unlikely that you have actually entered /0 in a line so the probable cause of this error is that a variable has been improperly handled by the program or an incorrect variable name used.

EXTRA IGNORED : An input statement expects a certain number of items eg INPUT A,B is looking for the input of two numbers. If you reply with more than that number of items, perhaps replying 10,12,14 then the program tells you that an item has been entered which will not be processed. The message will also be generated if you place commas into string inputs, since these will be interpreted as separators between items.

FILE NOT FOUND : You have either used the wrong filename or you have the wrong cassette/disc.

FILE NOT OPEN : You have tried to perform an operation on a file number which the 64 either does not recall an OPEN statement for or which has already been CLOSEd.

FILE OPEN : The opposite of the previous error. The most frequent cause is failure to include a CLOSE statement in some previous routine which used the same file number. Thus if you load data into a program using a file with '1' as its number but fail to CLOSE1 when the data is loaded, you will not subsequently be able to save data using file number 1.

FORMULA TOO COMPLEX : The simple answer may be to split the expression on the line in question into two expressions on separate lines. Unfortunately the error also crops up in a number of circumstances where the operating system becomes confused and in these cases no reliable guide can be given as to what the message means.

ILLEGAL QUANTITY : One of the variables used to access an array may be negative, or you may be trying to put a number outside the range

−32768 to +32767 into an integer array, or you may be trying to make a single byte function work on a number outside the range 0–255. If in doubt, try entering the various statements in the line in direct mode (without line numbers) and let the 64 tell you which one it doesn't like.

NEXT WITHOUT FOR : The program is not aware of the beginning of the loop the statement refers to. Either you have left out the FOR statement or you have jumped into the loop and the FOR has not therefore been executed.

NOT INPUT FILE : You have incorrectly specified an output file when you wanted to open a file to input data.

NOT OUTPUT FILE : Opposite of the above.

OUT OF DATA : You have tried to READ more data than there is in the program's data statements. See the section on data statements in Chapter 12 for a solution to this.

OUT OF MEMORY : There are four possible causes:

a) You have dimensioned arrays too big for the amount of available memory. For programs which will work with large quantities of data it is wise to use FRE to check on the available memory before finally deciding on the size of the arrays.

b) You have too many GOSUBs in operation at the same time, cluttering up the 'stack', the area of memory which has to remember their return addresses.

c) You have too many FOR loops operating, again cluttering up the stack, which has to remember the stage which each loop has reached. This may combine with b) if you have a long chain of GOSUBs *and* a large number of loops nested inside one another.

d) Unspecified. Sorry, but this is another error which can crop up if the 64 becomes confused.

OVERFLOW : Usually a problem with a variable being misdefined so that, for instance, a large number is being divided by a tiny fraction.

REDIMM'D ARRAY : You have tried to use the same array name in a DIM statement as that of another array already DIMensioned. If you wish to redimension an array you must first of all clear the memory.

REDO FROM START : An input statement was expecting to receive a number and was given a string. Note that an INPUT expecting a string will happily receive a number without any indication that something may be wrong.

RETURN WITHOUT GOSUB : You have entered a subroutine without the use of GOSUB. The most common cause is when subroutines are tagged onto the end of the program and a STOP statement is not placed at the end of the main part of the program.

STRING TOO LONG : In the process of string addition, input or input from a file you have tried to create a string whose length is greater than the 255 character overall maximum or the 80 character maximum for string input. The error also occurs when loading data which has not been properly separated by CHR$(13) when saved.

?SYNTAX ERROR : The 64 simply does not understand the line it is trying to execute. See the first part of the chapter for suggestions on how to track down obscure errors.

TYPE MISMATCH : Sometimes the context of the program dictates that a number be specified whereas what is actually encountered is a string (or vice versa). Thus A=A$+B$ would generate this error. The most frequent cause is careless omission of $ symbols in string handling lines.

UNDEF'D FUNCTION : You have tried to employ a user defined function but the program has no recollection of that function having been defined.

UNDEF'D STATEMENT : The line you want to GOTO or GOSUB does not exist. If the message appears to be nonsense at first sight, examine the number after the GOTO or GOSUB to ensure that you have not inadvertently substituted 'I' for '1' or 'O' for '0'. If this is not the reason then you may have tagged one line onto the end of the previous line as described in the first part of this chapter — the second part of the line appears to be there but isn't a line in its own right.

Conclusion

Successful debugging is something that comes with experience and no little thought and hard work. With the best will in the world there will always be some errors that will stump you, bugs that take days or weeks to find. When that time comes the most successful debugging aid is another mind on the subject, for very often another programmer will spot

something immediately that you were too close to the program to see. Even so, the time to call in someone else is not until you can describe exactly what is going wrong — the values of the variables involved, the likely area where the problem is cropping up. Simply knowing that a certain line generates a certain error is not enough, it is only the beginning of the process.

CHAPTER 3
Strings

Strings are an easy and flexible way of storing information on the 64. Using strings, information can be deleted, added or altered almost instantly, even in the case of quite complex operations. Playing around with strings can add quite considerably to what can be achieved by a programmer. This fact seems to be hidden from many, not because string handling is inherently difficult but because it is often fiddly and the lines expressing the string handling commands appear complex at first sight.

The 64 equips the user with three string handling functions, LEFT$, RIGHT$ and MID$. Most people understand their function fairly well but for those who are unclear a brief recap:

1) LEFT$(A$,10) means the *first* 10 characters of A$
2) RIGHT$(A$,10) means the *last* 10 characters of A$
3) MID$(A$,10) means the part of A$ which *begins with* character number 10 and continues to the end of the string.
4) MID$(A$,10,5) means the part of A$ which begins at character 10 and continues for five characters.

Applying these to an actual A$, which in this case will be the complete alphabet, the results are:

1) ABCDEFGHIJ
2) QRSTUVWXYZ
3) JKLMNOPQRSTUVWXYZ
4) JKLMN

On their own the commands are straightforward enough, as the examples show. The problem for most people seems to arise when they have to be applied in combinations. Once this happens lines tend to sprout brackets within brackets to a degree which makes their function appear very complex and obscure. When you run into this kind of problem the way to cope with it is always to begin with the part of the expression which is most embedded in brackets and then to begin translating the line so as to make it successively simpler. For example,

supposing you had an expression such as:

```
MID$(LEFT$(RIGHT$(A$,10),5),3)
```

What is to be made of it? Well let's assume that A$ is once again the alphabet. We begin with the string expression that is most embedded, ie RIGHT$(A$,10) because we don't have to translate anything else to get at the result. RIGHT$(A$,10) is 'QRSTUVWXYZ'. That leaves us with:

```
MID$(LEFT$('QRSTUVWXYZ',5),3)
```

Following the same rules we find that the LEFT$ part really means QRSTU, leaving us with:

```
MID$('QRSTU',3)
```

or STU. In string expressions as in any other kind of expression, start at the inside and work out and you will find that the problem solves itself.

In this chapter we shall examine ways in which the string functions can be combined, both with each other and with other BASIC commands, to provide a wide variety of interesting and useful programming capabilities. Several subsequent chapters will make use of the techniques described here in widely differing applications so it would be wise to ensure that you understand the examples given before moving on.

Concatenation or string addition

One of the simplest things that you can do with strings is to add them together:

```
100 A$=B$+C$+D$
```

would provide a new string consisting of the three strings on the right of the equation placed 'nose to tail' so to speak. This simple ability is often used to provide meaningful strings made up of smaller pieces of information. A simple example of this might be:

```
1000 REM*********************************
1001 REM STRING ADDITION
1002 REM*********************************
1010 INPUT 'SURNAME:';SN$
1020 INPUT 'FIRST NAME:';CN$
1030 INPUT 'SEX (M/F):';Q$
1040 SX$='MR.' : IF Q$='F' THEN SX$='MS.
```

```
1050 NAME$=SX$+' '+CN$+' '+SN$
1060 PRINT NAME$
```

Not every application of the technique need be so trivial, however. Individual items can be packed into a single string, separated by markers such as a '*' between each item, to save memory since every individual string stored has an overhead of three bytes in memory. Such simple 'packed' entries can be unpacked using a string search routine like that described later — more complex and flexible methods will be described in the chapters on Data Structures.

String subtraction

Just as strings can be added together, so parts can be subtracted from an individual string. This cannot be done quite as simply as addition since A$=B$−C$ would be meaningless to the 64. To subtract or remove one string from another simply means to redefine the original string so as to exclude the characters you wish to subtract. The precise method will differ according to the position of the characters to be removed in the main string:

1) To remove LL characters from the left hand end of A$, A$ must be redefined as the portion of A$ which comes after the first LL characters:

```
100 A$=MID$(A$,LL+1)
```

2) To remove LL characters from the end of A$, A$ must be redefined as all the characters up to and including the one before the first character to be removed. The way to accomplish this is to use LEN to discover how long the string is at the moment and then to say that it should now be that length minus the number of characters to be deleted:

```
100 A$=LEFT$(A$,LEN(A$)-LL)
```

3) To remove LL letters from the middle of a string it is necessary to know only where they start (SP) in addition to the length of the portion to be removed. Once this is known the string must be redefined as a concatenation of the portion *before* the characters to be deleted and the portion *after* them:

```
100 A$=LEFT$(A$,SP-1) + MID$(A$,SP+LL)
```

The logic of this line is that if the beginning of the group to be deleted is at character SP then we wish to keep all the characters up to and

including character SP–1. The actual group of letters is LL characters long so it will finish at position SP (the first character) + LL (the length) minus one. The second group of characters you wish to keep therefore starts at LL+SP in the string and continues to the end. An example of this would be subtracting CDE from the string ABCDEFG. The start position of CDE is character three, and the length is three characters. Thus the retained portions of the string would be up to SP–1, which would give AB and the characters after the group to be deleted would start at SP+LL, giving FG, adding together to give ABFG.

Inserting items into strings

Comparing what has been said about string addition and string subtraction you may notice that so far, while we can remove a group of characters from inside an existing string, we have not yet examined how to insert items into a string. In removing an item from within an existing string we examined a method of identifying the two remaining strings to be retained in the result. Roughly the same method is used in inserting a new group of characters. In the following example the object is to insert a new string, B$, into A$, with the first character of B$ becoming character PP of the amended A$:

```
100 A$=LEFT$(PP-1) + B$ + MID$(A$,PP)
```

Moving items within a string

Having examined how to add or remove items from a string we are now in a position to combine both techniques in order to move items around within a string. In essence, to move an item around in a string involves two operations: the group of characters to be moved must be subtracted from the string, then added again in another place. The method is illustrated below based on an original string A$, which has a group of characters, LL long, beginning at character position SP in the string. The object of the exercise is to move the group to a new position starting at character position FP. An example of such a task might be to rearrange the string ABGHICDEFJKL by moving GHI, which starts at character position three, to a position such that it will read ABCDEFGHIJKL. The new starting position of the group, in the final string, will be at character position seven.

Arriving at the correct position to reinsert the string is not completely straightforward. The first move is to ignore the fact that the group, in its original, is going to be deleted, and simply determine the new start position in the existing string. In the case of the example string above, we will want the reinserted group to start where the 'J' is at the moment, ie character position 10. We now have two alternatives:

a) If the new starting position of the group of characters is *before* their current start position then the figure arrived at needs no adjusting.

b) If the new starting position is *after* the current end of the group of characters then the length of the group must be subtracted from the figure arrived at.

In the example above the group ends at position five at the moment and the point at which we want to re-insert it is position ten. We have therefore to subtract the length of the group (3) to arrive at the final position, which is 7 as we found by commonsense above.

Putting this all together as it would apply to the example given above, we arrive at something like the following:

```
50 A$='ABGHICDEFJKL'
60 FP=10
70 SP=3
80 LL=3
2000 REM*********************************
2001 REM MOVE CHARACTER GROUP
2002 REM*********************************
2010 IF FP>(SP+LL-1) THEN FP=FP-LL
2020 TT$=MID$(A$,SP,LL)
2030 A$=LEFT$(A$,SP-1) + MID$(A$,SP+LL)
2040 A$=LEFT$(A$,FP-1) + TT$ + MID$(A$,F
P)
2050 PRINT A$
```

VARIABLES:
A$ String to be operated upon
FP Point at which character group would start if it were to be re-inserted without first having been deleted
LL Length of character group to be moved
SP Point at which character group to be moved currently starts
TT$ Temporary storage for character group to be moved

The lines beginning at 2000 can be used to move any group within the main string provided that the start position (SP), finish position (FP) and length (LL) are known.

Searching within strings

In all that has been said up to now about the manipulation of strings it has been assumed that the programmer already has all the information

necessary to specify start and finish points of all the sections of the string that are to be acted upon. Normally that is far from true and it will not be the programmer but the program which determines where changes are made according to its built-in guidelines. Very often the way in which the program will determine how to work on a particular string will be determined by the contents of the string itself.

Earlier in this chapter we examined an example of string addition using parts of full names. Each of the parts was stored in a separate string to begin with and then made into one longer string of the form 'MS. JANE SMITH'. Easy enough to do that way round but what about doing it in reverse, ie extract the parts from the whole? Getting MS. or MR. should be straightforward enough, since they are both the same length, but then we might find that we had one or two REV.s or other titles which aren't a standard length. Even if we could extract the title from the name with ease, the first name would still be unpredictable in its length. How then is the name to be disassembled?

The answer is that all the information needed to break down the name is contained within it in the form of the spaces which separate the three items. It is the spaces which we mentally use when the name is read and the same spaces can be used by the program, provided that we give it a method of searching the full string for the position of the group of characters which has to be operated upon. A simple method of searching a string, A$, for a particular combination of characters, TARGET$, would be as follows:

```
100 FOR I=1 TO LEN(A$)-LEN(TRGT$)+1
110 IF MID$(A$,I,LEN(TRGT$))=TRGT$ THEN
SP=I : GOTO 150
120 NEXT I
```

Here the routine will identify the start position of the first occurrence of TRGT$ (the character group being searched for) within the main string and store it in the variable SP. Using this kind of technique we can quickly design a routine which will unpack another string containing several units of information. In the example given below the three parts of a name given in the format 'MR. JOHN BROWN':

```
50 NAME$='MR. JOHN BROWN'
60 TRGT$=' '
70 DIM N2$(50)
3000 REM*********************************
3001 REM STRING SEARCH AND EXTRACT
3002 REM*********************************
3010 TT=LEN(NAME$)-LEN(TRGT$)+1
```

```
3020 S1=1
3030 IT=0
3040 FOR J=1 TO TT
3050 IF MID$(NAME$,J,LEN(TRGT$))<>TRGT$
THEN 3070
3060 N2$(IT)=MID$(NAME$,S1,J-S1) : S1=J+
LEN(TRGT$) : IT=IT+1
3070 NEXT J
3080 N2$(IT)=MID$(NAME$,S1) : IT=IT+1
3090 FOR I=0 TO IT-1
3100 PRINT N2$(I)
3110 NEXT I
```

VARIABLES:
IT Number of items discovered in NA$
N2$ Array used to hold items discovered in NA$
NA(ME)$ Main string to be searched
S1 Start of the section of NA$ to be searched
TR(GT)$ Character employed as separator between items
TT Last character in NA$ that it is worth comparing with TR$ without running off the end of NA$

The technique here is to start searching for the target string at character one in the main string and, every time the target string is found, to put the part of the main string from S1 up to the target string into the array N2$. The search is then recommenced at the character after the target string. The routine assumes that the main string does not end with the target string and so, after the loop is finished, takes the rightmost part of the main string which follows the last occurrence of the target string.

Such search techniques can be used in a variety of applications. Information can be packed into strings as shown in the section on string addition above and then a search routine used to extract the various parts again. In 'intelligent' programs which analyse the wording of instructions given to them, the routine can be used to test for the presence of certain verbs or nouns, a technique which is widely used in adventure games, for instance. In other cases the technique might be used in filing programs to detect entries in the file which contained a certain word.

Regular string structures

Because string handling can be accomplished by straightforward instructions which can either insert or delete portions without having to worry about having to move the existing contents, strings are an ideal place to store items of regular length where insertions and deletions have

to be made regularly. A single string with a maximum length of 255 characters can be used to hold 63 four-character data items, 50 of five characters and so on.

If everything is to have an identifiable place, the string must first be set up to the full length necessary to hold all the items. The easiest way to accomplish this is to use a loop to build up the string character by character. In the following routine, a string is set up capable of storing four-character items whose position can be specified by the user:

```
4000 REM**********************************
4001 REM STRING WITH SAME SIZE ITEMS
4002 REM**********************************
4010 A$='' : FOR I=1 TO 252 : A$=A$+' '
: NEXT
4020 INPUT 'ENTER FOUR CHARACTER ITEM:';
T$
4030 INPUT 'ENTER POSITION FOR ITEM (1-6
3):';T
4040 A$=LEFT$(A$,4*(T-1)) + T$ + MID$(A$
,T*4+1)
4050 PRINT [CLR];A$
4060 GOTO 4020
```

Data can be retrieved from a numbered location with equal ease. Add the following lines to the routine, enter some data and then try the second part by entering '****' when prompted for the string input:

```
4025 IF T$='****' THEN 4070
4070 INPUT 'NUMBER OF ITEM TO BE EXAMINE
D (1-63):';NN
4080 PRINT MID$(A$,NN*4-3,4)
4090 GOTO 4020
```

Items can be deleted from the array by simply redefining their positions as four spaces. These routines are not very robust, that is to say that they could be crashed easily by entering items which were not four characters in length. This, however, can be overcome with some simple error checks, as described in the next chapter.

Multiple element string arrays

One problem with using strings to store regular sized items of data is that of the maximum length of a single string. In the example given above it was accepted that the maximum number of items which could be stored

was 63. While this may be useful for a variety of applications, there are many others where a greater capacity would be desirable. This can be accomplished quite easily by declaring a string array and calculating the position of an item not only in terms of its place in the string but also of the correct place in the array as a whole. Given below is an adaptation of the previous routine which will accomplish this with a 20 element array, thus allowing 1260 four-character items to be stored and accessed:

```
50 DIM A$(19)
60 FOR I=1 TO 252 : A$(0)=A$(0)+'  ' : NE
XT
70 FOR I=1 TO 19 : A$(I)=A$(0) : NEXT
5000 REM*********************************
5001 REM MULTIPLE ELEMENT ARRAY
5002 REM*********************************
5010 INPUT 'ENTER FOUR CHARACTER ITEM:';
T$
5020 IF T$='****' THEN 5070
5030 INPUT 'ENTER POSITION FOR ITEM (1-1
260):';T
5040 LL=INT((T-1)/63) : T=T-63*LL
5050 A$(LL)=LEFT$(A$(LL),4*(T-1)) + T$ +
 MID$(A$(LL),T*4+1)
5060 GOTO 5010
5070 INPUT 'ITEM TO BE EXAMINED (1-1260)
:';NN
5080 LL=INT((NN-1)/63) :NN=NN-63*LL
5090 PRINT MID$(A$(LL),NN*4-3,4)
5100 GOTO 5010
```

VARIABLES:
LL The line number in which the new item must be placed, obtained by dividing the desired position by the length of the lines in the array

T/NN Originally number of item to be inserted or examined but transformed into position of item in line LL

Data in variable length strings

In the two previous sections it has been assumed that the strings being worked with were of a fixed length. The advantage of this is that the positions of items in the strings are fixed. There are always 63 items and an item placed at position 23 will remain at position 23 no matter what is done to other positions. Not all applications, however, require the full

length of the string in memory. Very often the need is to keep a compact list of items, whether in a particular order or not, which simply can be run through one by one, added to or deleted from. Given below is a routine which will allow four-character items to be added to the front of a list of up to 1240 items to be searched for or deleted:

```
6000 REM*********************************
6001 REM VARIABLE LENGTH STRING ARRAYS
6002 REM*********************************
6100 DIM A$(19)
6120 INPUT "1=INSERT/2=FIND/3=DELETE/4=S
TOP:";FF
6130 ON FF GOSUB 6200,6300,6400,6150
6140 GOTO 6120
6150 END
6200 REM*********************************
6201 REM INSERT
6202 REM*********************************
6210 INPUT "FOUR CHARCTER ITEM TO BE INS
ERTED:";IN$
6220 IF IT=1240 THEN PRINT "NO ROOM" : F
OR I=1 TO 2000 : NEXT : GOTO 6290
6230 A$(0)=IN$+A$(0) : IT=IT+1
6240 IF LEN(A$(0))<252 THEN 6290
6250 FOR I=0 TO 18
6260 TT$=RIGHT$(A$(I),4) : A$(I)=LEFT$(A
$(I),LEN(A$(I))-4)
6270 A$(I+1)=TT$+A$(I+1) : IF LEN(A$(I+1
)<252 THEN 6290
6280 NEXT I
6290 RETURN
6300 REM*********************************
6301 REM SEARCH
6302 REM*********************************
6310 INPUT "FOUR CHARACTER ITEM TO BE SE
ARCHED FOR";IN$
6320 FOR I=1 TO IT
6330 LL=INT((I-1)/62) : PP=4*(I-LL*62)-3
6340 IF MID$(A$(LL),PP,4)=IN$ THEN 6370
6350 NEXT I
6360 PRINT "NOT PRESENT" : FOR J=1 TO 20
00 : NEXT : GOTO 6390
6370 PRINT "PRESENT AT ITEM NO.";I : INP
```

```
UT "CONTINUE SEARCH:";Q$
6380 IF LEFT$(Q$,1)="Y" THEN 6350
6390 RETURN
6400 REM*********************************
6401 REM DELETE
6402 REM*********************************
6410 INPUT "FOUR CHARACTER ITEM TO DELET
E:";IN$
6420 GOSUB 6320 : IF I > IT THEN 6500
6430 A$(LL)=LEFT$(A$(LL),PP-1)+MID$(A$(L
L),PP+4)
6440 IT=IT-1
6450 PRINTIN$;" DELETED" : FOR I=1 TO 20
00 : NEXT
6460 FOR I=1 TO 18 : IF LEN(A$(I))=248 O
R A$(I)="" THEN 6490
6470 LN=248-LEN(A$(I) : A$(I)=A$(I)+LEFT
$(A$(I+1),LN)
6480 A$(I+1)=MID$(A$(I+1),LN+1)
6490 NEXT I
6500 RETURN
```

VARIABLES USED:
FF the number of the function chosen by the user
IT the number of four-character items in the array
TT$ temporary variable used to transfer an item to the next line when the length of one line in the array reaches 252, ie another item could not be added without exceeding the maximum length of 255
LL current line being dealt with in the array
PP position of item in line LL of the array
LN length of item to be moved down the array when deleting

Take some time to study this routine because, although it may seem a little daunting at first, there is little really new in it, all that has been done is to apply some of the techniques described in this chapter. The one novel feature is the way items are transferred from string to string if the length of a string reaches 252. This is done in order that, rather than testing first to see whether adding a new item would result in an illegal string length, the new item can always be added to the beginning of the array without danger. The reverse procedure is carried out when deletions are made.

Garbage collection

When using complex string routines which involve a great deal of moving strings around in arrays or redefining strings within the memory, you may every now and then notice that the 64 appears to pause. I say appears, in fact that is exactly what it does. When redefining a string the 64, or any other machine which uses Commodore BASIC 2 does not fully rearrange its memory to make the fullest use of the space available, strings are only moved about in the memory when absolutely necessary to make room for something which has increased in length. That means that redefining a string to be shorter does not free memory in the string memory area. The result is that when large amounts of string handling are being done the memory will gradually fill up. After a while the problem will become acute and the 64 will embark on what is known as 'garbage collection', or the tidying up of the memory so that only what is really needed is used for strings. Unfortunately, garbage collection does not always take place in time to prevent the machine stopping with the OUT OF MEMORY error message.

If you run into this problem in your programs the solution is to use the 64's FRE function, which forces garbage collection in order to provide an accurate value for the amount of free memory available. Thus:

```
PRINT FRE(0)
```

(the value in brackets makes no difference), will print the number of free bytes of memory. A slight complication is that FRE can only cope with numbers in the range –32768 to 32767. Anything over 32767 is expressed as 65536 minus the number of free bytes. Try the following program, entering everything exactly as it appears below with a space after the PRINT in line 15 but no space after the PRINT in line 20:

```
10 DIM A(1218)
15 PRINT '[CLR]'
20 PRINTFRE(0)
30 LIST
```

Run the program and you should see the value 32767 printed at the top of the screen, with the listing below. Go down to the listing and take out the space after PRINT in line 15. Because you have removed one byte from the program the amount of free memory should now be 32768 bytes, but running the program again results in the value –32768 being displayed. The reason for this is to do with the fact that the FRE function works with 16 bit integer arithmetic. Strictly this should mean that any

number from zero to 63535 can be expressed. In fact what happens is that the highest bit of the binary number is reserved for indicating whether the number is negative, so any number greater than 32767, where the 16 bit must be used, is interpreted as a minus number. The answer to the problem is to use a logical condition (see the separate chapter for an explanation) to convert negative numbers to the correct positive ones. The following line will always return the correct value:

```
MM=FRE(0) : PRINT MM-65536*(MM<0)
```

All this is a little bit of a digression. The real purpose of mentioning FRE is that to use it in any way during a program forces garbage collection, thus avoiding any danger of a false OUT OF MEMORY error. Simply include a line such as:

```
100 T=FRE(0)
```

at some point where it will be executed regularly. This will slow down the program a little, since garbage collection takes time and it should only be employed if experience shows it to be necessary.

Conclusion

Once you have made the techniques described in this chapter your own, you will never again need to despair when confronted with programs which make heavy use of string handling techniques, crammed with LEFT$, MID$ and RIGHT$ functions coupled with a mass of variables. Every string handling technique basically does no more than identify a part of a string and with intelligent use of variables there is very little that cannot be done to a string. Not only does this open up wide possibilities for more efficient data storage, it allows the programmer to write far slicker programs which allow the user to input strings in a more understandable format, leaving the program to do the work of identifying the important parts of what has been entered. As mentioned at the beginning of this chapter, the techniques described here will be drawn upon frequently in what follows, so don't skip the material here. It will more than repay the effort involved in understanding it fully.

CHAPTER 4
Inputting Information

One of the greatest differences between the micro-computers of today and the powerful mainframe computers of the past is that the modern micro is interactive, it responds immediately to the user and allows the user to work with a program as it is running. Before the micro, most people's experience of computers was of machines which had to be given both programs and all the necessary data first, without any tests being made of what was input, then the program was run. Very often the results of running the program would not be obtained for several hours, perhaps not until the following day, and errors in the program could mean that the whole process had to be repeated over and over again before the user even began to get an idea of what the program did when it was fully debugged.

The result was that the programmer, or at least the successful one, had to anticipate everything that was going to happen during the course of the program's execution. If there were going to be several different processes carried out during the course of the program they all had to be built in, in the correct order and with all the necessary data, before the program was submitted. If decisions had to be taken during the course of the program, they all had to be anticipated, for there was no way that the program could be designed to refer to the user for a decision. If an important decision had been overlooked then the program would have to be run again the following day.

The modern micro has changed that situation. Some applications of computers today, for instance games, would simply not have been possible to execute on machines which did not allow the program to refer back to the user as it was run. More importantly, users of serious applications programs now take it for granted that as the program is executed they will be able to feed in information, make decisions about the tasks to be performed, with the control of the program always in their hands. Such freedom, however, brings its own problems. Though it was often a painful process, non-interactive programming did ensure that people took care with their programs and with the choosing of the data for the programs to work on. Instant interactive computing has made us all careless. After all, the results of the program will be with us almost

immediately, so if anything has gone wrong it can be quickly put right and the program run again.

A good program today is not so much one where every item of data is spelled out in such a way that it will not cause a hiccup in the program, and every possible important decision anticipated. It is one which allows the user to input information in as flexible a way as possible and which refers to the user important decisions as to the way the program is to operate. Given the ease with which mistakes can be made during such a process, a good program is also one which ensures that the user does not unintentionally feed in material which causes the program to stop or corrupt its stored information.

In this chapter we look at some of the ways in which programs can accept information. In the next chapter we shall examine some of the methods by which the program can be protected against mistakes in that information.

Entering information: INPUT

The Commodore 64 provides two basic ways of entering information while a program is being run, the INPUT and GET statements. Both have their strengths though the majority of home-grown programs rely almost exclusively on INPUT, even in cases where it would be far more appropriate to make use of GET.

The main advantage of INPUT is that it is clear cut. A prompt appears on the screen, and letters or numbers can be entered and displayed. Almost as important, what is being entered can be edited using the cursor arrows in combination with the insert and delete keys. When the user is satisfied that what is on the screen is exactly what is desired, the INPUT is terminated by pressing RETURN.

INPUT does have disadvantages, however. Firstly it does place an upper limit of 80 characters on the information that can be entered (even to INPUT 80 characters you need to move the cursor down to the beginning of the line *after* the INPUT prompt, since only the contents of two consecutive screen lines can be accepted). Secondly there are several characters it cannot deal with properly, such as the comma. Thirdly the need to terminate every entry with a RETURN can be a real limitation in programs which require a large number of responses from the user or where you wish to continue with the execution of the program if the user makes no response. Despite these drawbacks the INPUT statement remains the workhorse of most programs that accept information while they are running.

Simple entries with INPUT

1) Entry of a single string:

```
10 INPUT 'ENTER A STRING'; A$
```

2) Entry of a single number:

```
10 INPUT 'ENTER A NUMBER'; A
```

3) Entry of several strings:

```
10 INPUT 'SURNAME, FIRST NAME AND SEX
   (SEPARATED BY COMMAS)'; SN$, CN$, SX$
```

```
Screen: SURNAME, FIRST NAME AND SEX
        (SEPARATED BY COMMAS)?

        LAWRENCE,DAVID,MALE <RETURN>
```

Note here how the commas in the information entered are used to identify the three separate strings the INPUT statement is expecting. If, instead of entering commas, the user presses RETURN after each item, the display would appear like this:

```
ENTER SURNAME, FIRST NAME AND SEX (SEPARATED
   BY COMMAS)? LAWRENCE

?? DAVID

??? MALE
```

4) Entry of several numbers:

```
10 INPUT "NUMERIC ITEMS 1-3:";A,B,C
```

This would behave in the same way as the string input in 3). Numbers and strings may be mixed in the same input line.

INPUT of several items using the same screen line

The flexible screen-handling of the 64, combined with the INPUT statement, can be used to reduce the clutter that so often spoils the appearance of the screen. When several inputs have to be made in

succession, for instance, and it is not essential that the user be able to see them all at once, it is a trivial matter to arrange all the input to fall on the same line, each overwriting the previous one.

1) Overwriting of INPUTs:

```
10 PRINT [10*CD]
20 INPUT '[CU] ITEM 1';A1$
25 INPUT '[CU] ITEM 2';A2$
30 INPUT '[CU] ITEM 3';A3$
```

In this case the PRINT statement at line 10 is simply an example which moves the print position to the desired screen location, which should be *one below* where you wish the INPUT prompts to appear. From then on all that is necessary is to put a 'cursor up' character at the beginning of each prompt and all the entries will be made on the same line. One problem which will arise with entries of different lengths is that the previous entry may not be fully erased by the subsequent ones. In that case, each INPUT line would consist of something like:

```
20 PRINT O$ : INPUT '[CU] ITEM 1';A1$
```

where O$ is a string of 39 spaces, preceded by a cursor-up character. Using this method, O$ would clear up to the end of the line and the input would always be placed on a clean line.

2) Entries can of course be spaced *across* the screen as well as down, provided you are sure that they will not overrun the line and spoil the format of the screen:

```
10 INPUT 'ITEM 1:';A$
20 PRINT [CU][10*CR];:INPUT 'ITEM 2:';B$
30 PRINT [CU][20*CR];:INPUT 'ITEM 3:';C$
```

INPUT to screen boxes

Few methods of entry look more professional than a screen with inverse boxes into which the entries are made. Though this is better tackled by the use of GET, it can also be done with INPUT if you are sure that the individual entries are not going to be longer than the boxes you allocate to them. The method would be something along the following lines:

```
10 PRINT '[9 SPACES] [RVS] [10 SPACES]'
20 INPUT '[CU] ITEM 1:[RVS]';A$
```

Line 10 would place a 10 space inverse box on the screen, with enough space for the intended prompt in non-inverse lettering in front of it. The INPUT prompt begins with a cursor-up to move back to the right line and the [RVS] at the end of the prompt ensures that whatever is entered will be in reversed lettering. Pressing RETURN at the end of the entry will cancel the RVS and subsequent printing will be in the normal mode.

It is worth taking the trouble to experiment with the many ways of laying out your INPUTs on the screen. A clearly formatted screen makes any program a great deal easier to use and reduces the number of careless errors that will be made in entries.

GET

Useful though INPUT is, there are many occasions where its limitations are irksome. In order to overcome this, 64 BASIC provides another command, GET. GET is harder to use than INPUT, since its sole function is to read the keyboard, that is to say detect whether or not a key is being depressed. GET does not wait for RETURN to be depressed before deciding what an entry finally is, each use of GET scans the keyboard and either accepts the first character it comes across or registers that no key is being depressed. GET is therefore particularly appropriate in cases where you wish to examine the length of an entry as it is being entered, where single key commands (without the use of RETURN) are desirable or where you do not wish the program to wait for an entry if the user is not depressing a key.

GET and creating a waiting state

The classic use of GET is to provide a waiting state until a single key is pressed:

```
10 GET A$ : IF A$='' THEN 10
```

On encountering this line the program will execute GET and determine whether or not any key is being depressed. If nothing is found, the string A$ will be given a null value and the IF statement in the second part of the line will cause the line to be re-executed. The program will thus wait indefinitely until a key is pressed. The moment a key is pressed, A$ will take on the value of the character that key represents and the program will pass on to the following line.

Slick applications programs which provide the user with choices at various stages during their execution will most often make use of this

form of GET. When a number of choices have to be made it is far easier to depress a single key that to constantly be having to press RETURN after every entry. Given below is a typical program menu which makes use of GET:

```
1000 PRINT '0 = QUIT PROGRAM'
1010 PRINT '1 = ENTER NEW DATA'
1020 PRINT '2 = DELETE ITEMS'
1030 PRINT '3 = ALTER ITEMS'
1040 PRINT '4 = DISPLAY DATA'
1050 PRINT '[CD] WHICH DO YOU REQUIRE?';
1060 GET IN$ : IF IN$='' THEN 1060
1070 TT=VAL(A$)
1080 ON TT GOSUB 10000,2000,3000,4000,50
00
1090 GOTO 1000
```

Here all that is required is that the user touch 0, 1, 2, 3, or 4 in order to call up that part of the program.

Moving a cursor with GET

Games programs too make extensive use of GET, especially when it comes to moving objects around a screen:

```
1000 REM*********************************
1001 REM MOVING CURSOR
1002 REM*********************************
1010 GET T$
1020 PRINT '* [CL]';
1030 FOR I=1 TO 50 : NEXT
1040 PRINT '[SPACE][CL]';
1050 FOR I=1 TO 50 : NEXT
1060 IF T$='' THEN 1000
1070 IF T$+'[CU]' OR T$='[CD]' OR T$='[C
L]' OR T$='[CR]' THEN PRINT T$;
1080 GOTO 1000
```

This routine provides a simple flashing '*' cursor which will flash on the screen until a key is pressed. The two loops provide a short pause so that the cursor appears to flash rather than flicker. To move the cursor, all that you need to do is to examine the character placed into T$ by the GET statement and, if it is one of the cursor move characters, to PRINT it.

The print position will move and you can then re-execute the flashing cursor routine.

GET and timed responses

In games and sometimes in more serious applications it can be useful to allow the user a certain amount of time to make a response, rather than stopping the program indefinitely. Using GET and a suitably sized loop, this is a simple matter:

```
10 FOR I=1 TO 1000
20 GET T$ : IF T$<>'' THEN I=1000
30 NEXT I
40 IF T$<>'' THEN GOSUB 500
```

Here the user has the time it will take to execute the loop 1000 times in which to respond. If a response is made then a particular course of action can be executed, as in line 40. If no key has been depressed during the 1000 iterations of the loop, the program continues at the line following line 40.

GET and inverse boxes

We have already seen that an effective way of entering data is to set aside a specific position for it, say an inverse box. In this kind of application GET can be vital because it can be used to detect how long an entry is while it is still being made. The following routine will accept any entry up to and including 10 characters in length:

```
2000 REM**********************************
2001 REM ENTRY TO INVERSE BOX WITH GET
2002 REM**********************************
2010 IN$=''
2020 PRINT '[7 SPACES][RVS][10 SPACES]'
2030 PRINT '[CU]ITEM 1:[RVS]';
2040 GET T$ : IF T$='' THEN 2040
2050 IF T$=CHR$(13) THEN 2110
2060 IF T$=CHR$(20) AND LEN(IN$)=0 THEN
2040
2070 IF T$=CHR$(20) THEN IN$=LEFT$(IN$,L
EN(IN$)-1) : PRINT '[CL][SPACE][CL]'; :
GOTO 2040
2080 PRINT T$;
2090 IN$=IN$+T$
```

```
2100 IF LEN(IN$)<10 THEN 2040
2110 STOP
```

Here the box is printed and the prompt placed in front of it, leaving the print position at the beginning of the box. Letters can then be typed into the box. If the DEL key is pressed (CHR$(20)), provided that there is already something in the box, the last letter of what has been input is erased, both from the screen and from the memory — you cannot use the cursor move arrows as the routine stands, only DEL. Entry is terminated either when RETURN (CHR$(13)) is pressed, or when the entry reaches a length of 10 characters.

Using a routine such as this you can prevent an entry being longer than you desire and thus ensure that the desired layout of the screen is not spoiled or the structure of arrays corrupted.

Screen editing with GET

GET can also be used both to overcome the limit of length of strings which is imposed by INPUT and to edit existing strings, something that is very difficult with INPUT. The technique of changing things as they appear on the screen and being able to record those changes is called screen editing. The 64 allows you to perform screen editing on program listings by moving the cursor over existing lines and adding to or subtracting from them. The same techniques can be applied to strings on the screen, though such applications are seldom simple.

The following routine is fairly complex but its purpose is a simple one. What is does is to allow the user to enter a string (called A$) and to edit it as it is entered. If A$ began as a string which was already in memory, then the routine could also be used to print it to the screen and change it. The only limitation is the absolute one imposed by the maximum 255 character length of a string.

```
3000 REM**********************************
3001 REM SCREEN EDITING
3002 REM**********************************
3010 DEF FNA(P)=1024+LL*40+P
3020 A$='[1*SPACE]'
3030 P=0 : LL=19
3040 PRINT '[HOME][18*CD]'
3050 PRINT A$
3060 CH=PEEK(FNA(P)) : POKE 54272+FNA(P)
,14 : POKE FNA(P),160
3070 FOR TT=1 TO 5 : NEXT TT : POKE FNA(
P),CH
```

```
3080 GET T$ : IF T$='' THEN 3040
3090 IF T$=CHR$(13) OR LEN(A$)=255 THEN
STOP
3100 IF T$=CHR$(95) AND P<>0 THEN 3030
3110 IF T$=CHR$(95) AND P=0 THEN P=LEN(A
$)-1 : GOTO 3040
3120 IF P>0 AND T$=CHR$(20) THEN A$=LEFT
$(A$,P-1)+MID$(A$,P+1): P=P-1
3130 IF T$=CHR$(20) THEN 3040
3140 IF T$='[CU]' OR T$='[CD]' THEN 3040
3150 IF T$<>'[CL]' AND T$<>'[CR]' THEN A
$=LEFT$(A$,P)+T$+MID$(A$,P+1) : P=P+1
3160 IF T$='[CL]' AND P>0 THEN P=P-1
3170 IF T$='[CR]' AND P<LEN(A$)-1 THEN P
=P+1
3180 GOTO 3040
```

Unpacking the routine, what we find is this:

3000 This is an important function when it comes to screen editing. What it does is to PEEK a particular character position on the screen. We need such a function in order to be able to detect later what is already on the screen and remember it in case we wish to replace it in the same position. User-defined functions are described more fully in Chapter 12.

3020 In order to start on the process of editing a string, we need a string to edit. This line assumes that we are starting from scratch and begins our string as a single space. If the intention was to edit an existing string then it would be temporarily renamed A$ and this line would be used to add a space to the end of the string.

3030 The variable P is used during the routine to record the position of the cursor in the string, starting the numbering at zero.

3040-3050 This line prints the desired string on a line dictated by the number of 'cursor downs'.

3060 This line uses the defined function to discover the screen-code value of the character at position P in line LL and store it in the variable CH. Into its place is POKEd an inverse space.

3070 The time taken to execute the small loop keeps the inverse space on the screen for a fraction of a second, then the original character is replaced

on the screen.

3080 Having flashed the cursor, this line checks to see whether a key is is being depressed. If not, the cursor is flashed again.

3090 To reach this part of the routine, the user must have depressed a key. This line tests to see whether the key depressed was RETURN or whether the string being edited is already 255 characters long. In either case the routine is terminated. In actual use you would not use a STOP but would GOTO another part of the program which would make use of the newly created string. Before using the string, you would strip off the last character, the space that was added at the beginning.

3100 If the character entered is the 'left arrow' at the top left hand of the keyboard, then the print position (P) is reset to the beginning of the line — provided that it is not already there.

3110 If the left arrow has been entered and the print position is already at the beginning of the line, then the print position jumps to the end of the line. These two lines are added simply to make it easier to move around the string.

3120-3130 Provided that the print position is not at the beginning of the line, pressing the delete key will remove the character to the left of the flashing cursor. The program then returns to the flashing cursor routine.

3140-3150 If the character entered is not one of the cursor move arrows, then the character is added into the string at the cursor position, with all characters from the cursor position onwards being moved one space to the right to make use of it. The string is then reprinted.

3160-3180 These lines test to see whether the character entered was either the cursor left or right arrow. In this case there is no need to reprint the string. All that is altered is the position at which the cursor will be flashed.

Using this routine and a little imagination you will be able to place information onto any position on the screen, edit it using the left and right cursor arrows, add or delete characters and then replace the edited information into memory. You can, if you wish, add another variable to record the position of the cursor down the screen as well as across. This allows you to move up and down between different lines of text with the program noting which of the lines is being edited at any one time. This kind of method is far easier on the user than having to recall a string for examination and then re-entering it in full in its altered form. For

programs where information has to be regularly updated and altered a routine of this type can make the difference between a cumbersome program and one which is a joy to use.

Simple screen-editing using INPUT

For all the flexibility given to us by a routine such as that above, it is still possible to perform screen-editing using INPUT statements, though you will not be able to input more than 80 characters in this way and can only deal with one string at a time. To achieve this, simply print the string or number to be edited on the screen with enough room for the input prompt in front of it:

```
100 A$='ABCDEGF'
110 PRINT TAB(9);A$
120 INPUT 'CU]ITEM 1:';A$
```

With this method you can use all the automatic screen editing facilities provided by the 64's operating system and, when you press RETURN, the 64 will accept whatever is after the prompt as the new form of A$.

Conclusion

The input of information is one area where it is very easy to spoil an otherwise excellent program. Information which is requested on a cluttered screen, with unclear prompts and in no apparent order will often be entered wrongly and will certainly be tiring to enter. Proper formatting will make all the difference, as will the added use of colour, with different colours attached to successive prompts, for instance, but with each prompt ending in a black colour control character which will make the responses stand out on the screen from the prompts themselves.

Which method or format you prefer will be a matter of personal choice but with the techniques outlined in this chapter there is no reason why your program need ever again be limited to boring lists of single colour prompts, one after another on the screen. It remains to be seen how you can be sure that the information you obtain with your prompts is correct.

CHAPTER 5
Error Trapping

There is no such thing as an idiot-proof program. The best that can be said about any program is that it has not yet come against a creative enough idiot. Even so, there are few things more annoying than an attractive and useful program which stops at the crucial point because the user makes an input which cannot be dealt with in the normal way. In this chapter we shall examine some of the ways in which a program can be made more robust, that is to say less likely to stop in confusion when unexpected or ridiculous data is input. You will find very few complex techniques in the chapter, for error trapping is largely a matter of common sense, an attempt to understand the kind of mistakes people are likely to make and to be one step ahead of them.

Avoiding errors — common sense precautions

Why do people insist on making mistakes when using your best programs? The answers are many but the largest proportion of them boil down to the fact that your favourite program is not quite as good as you think. Most input errors arise because the program is simply not clear enough in the way that information is requested from the user. It may be that the prompts you give with your INPUT statements are too brief, or that the screen is too cluttered to be able to concentrate properly on each prompt, it may be that you change the conventions on which you work halfway through a program, perhaps expecting a numeric response for most of the time then suddenly asking for an alphabetic input without making it clear. Whatever the reason, there is no point in lamenting the limitations of the people who crash your programs, the programs are meant to be used and that will only be possible if they are designed in such a way as to make clear to the user exactly what is meant to be happening from moment to moment.

With this in mind we can start with one or two commonsense principles which will eliminate the vast majority of errors:

1) Take the trouble to format the screen properly using the kind of techniques described in the last chapter. The layout of the screen should be such as to draw the eye of the user to the correct prompt and away

from anything which might distract. Avoid clutter on the screen.

2) Try to get someone else to look at the prompts you are using before you decide the program is finished. Having spent days or weeks developing the program no doubt you know the purpose of every entry of information backwards. Other users will have only the haziest idea what the program is about and no idea what each input is for unless it is spelled out. This applies even if you do not intend to let anyone else get their hand on the program. If you leave the program for a time and then come back to it you may well find yourself as puzzled as anyone else.

3) Specify the format of the entry wherever it is in doubt — is it in figures or letters, is there a maximum value, is there a limit on the length in characters, should there be commas between items....? Take the example of a menu which appears on the screen:

```
0 - Quit Program
1 - Enter New Item
2 - Delete Item
3 - Search For Item
4 - Display Data

Which do you require?
```

That seems fairly clear yet you can be sure that some people will sit there entering the letters 'Display Data' and wondering why nothing happens or the program stops. The prompt really should have been 'Which number do you require?'. In other words if the program assumes the entry will be made in a certain way, it should first tell the user.

4) Do not make combinations of inputs too lengthy or complex. If you really need to input 10 items in a row then break them up visually on the screen into logical groups, perhaps clearing the screen between the groups. Paradoxically you will get more errors with complex inputs when the user has become more familiar with the program since less care will be exercised in reading the prompts, the fact that someone uses the program correctly the first few times does not mean that they won't start to mentally freewheel and respond to the wrong prompt when they are more experienced with the program.

5) Be sure you use the same conventions throughout the program. If an input of '0' normally quits the current function but in one place deletes an item of data, don't be surprised to find that you are not very popular with someone who loses something important when all they wanted to

do was return to the menu.

6) Be extremely careful with inputs whose order sometimes changes. If you normally ask for name, address and age in a filing program, but need to know another fact if age is greater than 65, don't just put the extra prompt on the screen without warning. The user will have been accustomed to making three inputs and it is odds on that he or she will press RETURN again when the data does not immediately disappear from the screen after the third input. If you are changing the order that the user has become accustomed to, flash the screen a different colour or make a beep to remind them that something different is happening.

7) It is a good idea to assume that, despite your best efforts, mistakes have been made through tiredness, boredom or sheer carelessness. Such mistakes can, however, be reduced by printing out the information the user has just entered in a new format and asking for confirmation that this is correct. It may just be that the user will then notice that age and telephone number have been transposed, for instance. Echoing an input to the screen implies that nothing is actually done with the input until the user has confirmed that the information is as intended. Placing an input straight into the main array of data makes it very difficult to remedy the situation if the user specifies that a mistake has been made.

Error Trapping — some simple programming techniques

The use of common sense methods like those described above will reduce the level of errors made out of all proportion to the effort involved. Errors will still be made, however, and we shall now examine some of the ways in which the program can be made proof against them by building checks and balances into the program.

Setting limits

One of the commonest sources of program crashes occurs when the program is designed to work with data which falls within certain limits and an item is input which falls outside those limits. For example, if you write a program to input two numbers and then divide the first by the second, then the program will stop with an error if the second number input is a zero. Clearly the prompt should have instructed the user to input a number greater than zero, but even when this is done there is still the possibility of a typing error or a more than ordinarily obtuse user. Where a program can only function within limits it is wise to put in some simple checks that any information entered does not fall outside. The limits

which usually cause problems are value of a number, and the length of a string.

1) Value of a number

In developing and debugging your program, one of the first things you should have determined is the range of values for numerical inputs that the program would tolerate. In the example given above, zero is a clear case of an invalid input. If such limits are found they can be simply built into an error-checking line. Suppose, for instance, that a particular input needed to be a number from one to ten inclusive:

```
1000 INPUT 'ENTER A NUMBER IN THE RANGE
1-10';NN
1010 IF NN>=1 AND NN<=10 THEN 1050
1020 PRINT 'THAT NUMBER IS OUTSIDE THE R
ANGE!'
1030 FOR I=1 TO 2000:NEXT I:PRINT '[2*CU
]';O$:PRINT O$:PRINT [2*CU]';
1040 GOTO 1000
```

All that routine does is to check that the input is within the specified limits and, if it is not, displays an error message for the duration of the loop. Error message and original input are then erased by printing O$, a 39 character string of spaces, twice and the user can try again. Printing one long string which covers both lines will not work since the 64 will remember that the line-end has been crossed and will treat the next INPUT as if two lines of characters had been received, thus ruining the format by shifting the print position too far down. Note that although giving a message to the user and then clearing the original input takes more lines than simply going back to the original INPUT statement if the number is wrong, it is worth the effort because it tells the user *why* the input was rejected. There are few things more frustrating to a user than to have an input rejected without understanding why.

2) Length of a string

Just as the value of a number can be out of the desired range, so some programs can be thrown by the fact that they are expecting a string of a certain length and do not get it. Guarding against this is little different from the technique employed for numbers:

```
1000 INPUT 'ENTER A STRING (LENGTH 1-10)
';A$
```

```
1010 IF LEN(A$)>=1 AND LEN(A$)<=10 THEN
1050
1020 PRINT 'THAT STRING IS THE WRONG LEN
GTH!'
1030 FOR I=1 TO 2000 : NEXT I : PRINT '[
2*CU]';O$:PRINT O$:PRINT '[2*CU]';
1040 GOTO 1000
```

Here the check is that the length of the string falls in the range 1-10 and the lines correspond exactly to the lines of the number checking technique above. One extra safeguard you might like to include when inputting strings is an extra command before the INPUT statement:

```
990 A$=''
```

This takes care of the fact that pressing RETURN without making an input will leave A$ as it was before the INPUT. Thus if A$ already has a value of 'SMITH', then pressing RETURN by mistake would have left it as 'SMITH' and the length check would have been evaded.

Garbled entries

Often, either through a simple typing error or through failure to read the prompt correctly, the user will make an input which is not so much out of range as incomprehensible to the program. In the case of numeric inputs this usually happens when a letter is inadvertently entered instead of a number. In the case of strings it happens when the program has a list of strings it understands, perhaps as commands, and the string input does not correspond to any of them.

1) Invalid number formats

This type of error occurs when, for instance, instead of responding to a prompt for 'age' the user carelessly inputs a string, such as part of an address. The result, of course, is nonsense, with the input usually having a value of zero to the 64 (unless it begins with one or more figures). The only way to guard against such mistakes effectively is to have numbers input as strings and then examined character by character to see that every character is a valid digit. A typical routine would be as follows:

```
1000 NN$=''
1010 INPUT 'ENTER NUMBER:';NN$
1020 IF NN$='' THEN NN=0 : GOTO 1080
```

```
1030 FOR I=1 TO LEN(NN$)
1040 IF MID$(NN$,I,1)>='0' AND MID$(NN$,
I,1)<='9' THEN 1070
1050 PRINT 'THAT IS NOT A VALID NUMBER'
: FOR J=1 TO 2000 : NEXT J
1060 PRINT '[2*CU]';O$ : PRINT O$ : PRIN
T '[2*CU]'; : GOTO 1010
1070 NEXT I
```

Such a technique does not add noticeably to the time taken to process an input and will be effective in detecting errors which would otherwise not be notified to the user. Invalid number formats do not result in the program crashing, they simply mean that unintended values are extracted from what has been input.

Note that in the routine as given there is no check against the user simply pressing RETURN, it is simply that such an input is interpreted as zero. If you wished to guard against such an eventuality you would add an extra message, perhaps saying NOTHING INPUT.

2) Unknown strings

Where a program is designed to accept a command consisting of one of a number of strings, confusion can arise if a command is mistyped on entry. Take for example a program which allows data to be input for any one of the months of the year. The user could be allowed to input the month number but this is prone to error. You might therefore decide to allow the user to input the actual name of the month for which the input is to be made. In that case there would need to be a check that the input was a sensible one:

```
1000 MM$=''
1010 INPUT 'NAME OF MONTH:';MM$
1020 FOR I=0 TO 11
1030 IF MM$=MONTH$(I) THEN 1070
1040 NEXT I
1050 PRINT 'MONTH NAME INVALID' : FOR I=
1 TO 2000 : NEXT
1060 PRINT '[2*CU]';O$ : PRINT O$ : PRIN
T '[2*CU]'; : GOTO 1010
```

Here the assumption is that you have stored the names of the months in elements 0-11 of the array MONTH$ when the program first commenced. The routine checks what has been input against the recorded month names and, if no match is found, prints out the error message.

Common sense error-trapping: 'the second look'

In the section of common sense measures given above we noted that one of the most effective methods of reducing the level of errors on input is to force the user to look again at what has been entered and confirm that it is correct. If you wish to do this regularly during the course of a program the task can be handed over to a subroutine which will save space and at the same time ensure that the presentation is uniform throughout the program. An example of such a subroutine is given below:

```
1000 PRINT LEFT$(CC$,LL+1);
1010 FOR I=0 TO NQ-1
1020 PRINT '[GREEN]';PP$(I);':'; : INPUT
 '[BLACK]';QQ$(I)
1030 NEXT I
1040 GOSUB 1120
1050 PRINT LEFT$(CC$,LL+1);
1060 FOR I=0 TO NQ-1
1070 PRINT '[YEL]';PP$(I);':[RVS]';TAB(2
0);QQ$(I)
1080 NEXT I
1090 INPUT '[CD]ARE THESE CORRECT (Y/N):
';TT$
1100 GOSUB 1120 : IF LEFT$(TT$,1)<>'Y' T
HEN 1000
1110 RETURN
1120 PRINT LEFT$(CC$,LL+1);
1130 FOR I=1 TO NQ+2 : PRINT O$ :NEXT
1140 RETURN
```

Variables required:
LL — screen line position of first prompt
NQ — Number of questions
PP$ — Array containing the prompts

What the routine accomplishes is to ask a series of questions based on the variable NQ. The questions themselves are stored in the array PP$ before the subroutine is called. Responses are stored in QQ$. When all the questions have been asked, they are redisplayed, with the answers in inverse. If the user does not answer Y to the question ARE THESE CORRECT, the questions and the answers are cleared and presented again.

Formatting of the screen is accomplished by the use of a string called CC$ which consists of:

```
'[HOME] [24*CD]'
```

and the correct area of the screen is cleared by use of the usual O$, consisting of 39 spaces.

The actual format of the prompts and the way that they are presented again to the user is a matter of taste, the only point of importance is that changing their appearance helps the user to concentrate on them for the second time. At the end of the routine, if you do not wish to leave the information on the screen, the subroutine at 1200 could be called to clear the relevant lines.

To call the routine you would have lines something like the following in your main program:

```
500 LL=10 : NQ=3
510 PP$(0)='NAME'
520 PP$(1)='ADDRESS'
530 PP$(2)='PHONE'
540 GOSUB 1000
550 FOR I=1 TO NQ : AA$(I)=QQ$(I) : NEXT
```

Once the main prompt routine above had been entered, these brief lines would specify the position of the first prompt on the screen (in lines), the number of questions to be asked and then spell out the prompts. Having done that the subroutine would format the prompts and ask the user for verification of the accuracy of the responses, repeating the prompts until the responses are made correctly. On return from the subroutine the responses are contained in the array QQ$ and can be transferred to wherever you wish to permanently store them.

Clearly there would be no point in using a routine like this one in a small program or one where there were only one or two prompts ever used. For a program which asks quite a few questions, however, such a subroutine can increase the accuracy of the responses while at the same time reducing the need to include the checking routine every time a prompt is used.

DIY error messages: the elegant solution

So far we have been considering the possibilities of routines which can be slotted into various parts of the program to provide a variety of error traps, mainly involving incorrect inputs. All of this depends on being able always to determine what constitutes an incorrect input as soon as it is made. That will not always be possible. Sometimes an input will be made

that is not too long, or too low in value, that cannot be easily checked for misspelling and yet it will still be an input that will cause problems for the program. The difficulty here is that the actual difficulty created will not become apparent until the information which has been input has been at least partially processed by the program. With enough foresight it would be possible to anticipate where such errors might crop up in the program and place checks and error messages at the correct points.

Problems still remain, however. Suppose that a certain program accepts an input and then passes control to five successive subroutines, each with a distinct task to perform and ending with the processed data being stored permanently or used to modify the overall result of the program so far. Now suppose further that in the fourth of the five subroutines it becomes clear that there is something wrong with the data that has been input, not something that would actually crash the program but nevertheless some fault that would make the final result of processing the information yield a nonsense. What is to be done?

Clearly we could insert an error check in the fourth subroutine, with an appropriate error message to be displayed. But we also have the problem of how to escape from the chain of five subroutines. It is no use simply terminating the fourth subroutine since that will normally result in the fifth subroutine being executed and we have already decided that that would be disastrous. What we must do is to make a record of the fact that an error has been found and use that record to ensure that no more work is done on the particular data. So on returning from subroutine four we would place a special check in the program to ensure that subroutine five would not be executed if the particular error had been discovered. This would work but in a program of any complexity it is possible that there could be many places where errors could arise and we could end up with a whole range of special checks, each one ensuring that one particular error has not arisen.

The elegant solution to the problem is to hand over the recording of any errors that are found during the course of a program to a single variable which will be tested regularly, and to use one subroutine to handle all the error messages that are needed. Using this method it is easy to insert new error checks and to define new errors, while problems of how to prevent different parts of the program executing if a problem has arisen will disappear.

As an example of the use of this technique, consider a program where three common errors tend to crop up:

1) After a certain amount of calculation, a figure is generated which is too large for the program to cope with sensibly.
2) A label is entered for data that is already contained in the program's memory.

3) The data must be entered in a fairly complex format and mistakes in this can only be checked after the data has been partially processed.

You would begin by defining, when the program was first initialised, an array called ERR$(3) and you would then set three elements, one to three, of the array to:

a) RESULTING NUMBER TOO LARGE
b) LABEL DUPLICATION
c) FORMAT ERROR ON INPUT

In addition to this we would declare a variable called ERR and set it equal to zero.

In the program itself, the checks for specific errors and the error messages associated with them are replaced by checks which do nothing more than alter the value of ERR when an error is found. For instance:

```
1520 IF NN>65535 THEN PRINT 'NUMBER TOO
     LARGE'
```

would be replaced by:

```
1520 IF NN>65535 THEN ERR=1
```

Into each relevant subroutine of the program we would now place a new line which would ensure that if any error had been detected, the subroutine would not operate. Thus the entry line of a subroutine might be:

```
1700 IF ERR <> 0 THEN RETURN
```

This would ensure that if the value of ERR were anything but zero, the subroutine would not be executed. Subroutines which call other subroutines might have their calls to other routines altered so that whenever the program returned from a subroutine with an error being indicated, nothing more was done to the data:

```
1770 GOSUB 2500 : IF ERR <>0 THEN RETURN
```

In the end, program execution would return to the main module which dictated the order in which subroutines were called. In that module we would insert a line something like this:

```
1120 IF ERR THEN GOSUB 3000
```

which would call a simple subroutine consisting of:

```
3000 REM*******************************
3001 REM ERROR MESSAGES
3002 REM*******************************
3010 PRINT ERR$(ERR)
3020 ERR=0
3030 RETURN
```

The beauty of such a system is that if you decide you want to protect against any new errors as you develop the program then all you have to do is to add a new error message to ERR$, insert a check which sets ERR to the appropriate value and the built-in lines which test whether ERR is zero will ensure that the program is now protected against the possible error you have identified. You can go on identifying new errors to your heart's content with very little effort.

Conclusion

Using the techniques laid out in this chapter you will be able to produce programs which will survive most of the abuse that users will subject them to. How far you want to go in protecting your program will depend partly on how complex it is and partly on how important is the data that it works upon. Even a simple program may work on information which takes a long time to enter, causing severe inconvenience when it crashes after half an hour's work. Perhaps the most important factor, however, is the satisfaction which comes from having designed a program which appears to be in control of events rather than one that has to be handled gently for fear that it will stop in confusion.

Even so, be warned! Never boast of how robust your favourite program is. The person you are boasting to will inevitably press a single key you hadn't thought of and your reputation will be shot down in flames.

CHAPTER 6
Storing and Retrieving

One day we shall all be using a generation of computers with memories so large that they will hold, at one and the same time, all the information and programs that we wish to use. Not only that, the information and programs will be a permanent part of the memory, always available as soon as the machine is switched on. The possibilities opened up by such machines will represent as big a step forward as the home micro-computer has been over the past few years. Until that day, however, micro owners have to learn to live with machines capable of holding only a part of the total sum of information that may be used from day to day.

To cope with the fact that a machine like the 64 can hold only one program and the data associated with it at any one time, Commodore provide an extension to the memory of the 64. That is what the C2N Cassette Recorder, or the 1541 Disk Drive, really is — extra memory. Admittedly, compared to the speed with which data can be recovered from the RAM chips of the 64, the cassette recorder and even the disk drive work at a snail's pace. Effective programming, however, especially programming for most useful applications, will learn to make use of the extra capacity of the C2N and 1541 to overcome both the size limitations of the 64's memory and the fact that the present generation of micros cannot hold information during the time the machine is switched off.

Strangely, though most micro owners seem prepared to pay large amounts of money for add-ons which will increase the memory capacity of their machines, often by only small amounts, few home grown programs make much use of the massive storage capacity of a floppy disk or even a humble cassette. In this chapter we shall examine some of the techniques necessary to make better use of the C2N and 1541. The chapter, especially as it relates to the use of the disk drive, is not exhaustive, indeed it would be quite possible to write a complete book on the effective use of floppy disks. Even so, using the techniques described in these pages you will be able to store information more reliably and in larger quantities, swapping information with that stored on tape or disk and making better use of the memory available.

Saving programs

The first and most obvious use of external storage is to keep your programs safely for future use. It is always surprising how little care most people take in this, failing to save regular updates when a program is being developed, failing to check that a program has been properly saved, keeping only one copy of important programs and abusing tapes and disks by leaving them around exposed to the elements. Given below are one or two commonsense rules when it comes to saving programs.

1) As you develop new programs SAVE them regularly. Like any other microcomputer the 64 can lose programs if there is a momentary surge in the electricity supply, or someone kicks the plug or even because in your programming you manage to upset the 64's equilibrium. How much work you will have lost will depend on how long it has been since you last saved your program. If a program is being entered rapidly I would not normally expect to enter lines for more than 15 minutes without resaving the program. In the event that I am debugging a program, so that relatively fewer changes are being made, I might increase that period to half-an-hour. It really depends on how much you are prepared to lose but you can depend on the fact that if you do not save programs regularly you will, sooner or later, lose an important program that has taken a long time to enter.

2) To make saving a program easier and to encourage myself to do it, I always include four lines at the beginning of my programs which allow saving without having to spell out the program name each time:

```
1 GOTO 4
2 SAVE 'PROGRAM NAME' : INPUT 'REWIND TH
EN ANY KEY TO VERIFY';Q$
3 VERIFY 'PROGRAM NAME' : STOP
4 REM
```

Including such a routine in a program has the virtue that you are unlikely to save the program under the wrong name due to a typing error, it can be saved simply by entering 'GOTO 2' and, as an added bonus, it means that all your programs can be started with a uniform 'GOTO 1' if you do not wish to use RUN and wipe out any stored variables. The use of VERIFY is optional and many people never use it, but see the next point in this section before making a choice. If you do decide to dispense with VERIFY you can of course lose line 3 but you will still need the STOP or the program will begin to execute every time you SAVE it.

Those using a disk drive can include a similar routine to that given above:

```
1 GOTO 3
2 SAVE '@0:PROGRAM NAME',8 : VERIFY 'PRO
GRAM NAME',8 : STOP
3 REM
```

A point of caution here for disk owners is that due to a disk operating system bug, on relatively full disks the 1541 drive may sometimes corrupt its own map of the disk when using the '@O' prefix to the file-name (this ensures that a program is saved even though there is an existing file of that name). The program just saved will be unharmed but you may find it difficult to load another program on the disk. The solution, if this turns out to be a problem, is either to first scratch the existing program file and then resave the new version, or to attach a number to the end of the program name in line 2 and change it each time the program is saved, or to leave the name alone and validate the disk after saving.

In the case of disk drive owners there is never any excuse for failing to VERIFY a program — if you don't then you deserve everything you will undoubtedly get.

3) Loading and saving on the 64's C2N cassette recorder are among the most reliable on any machine on the market today. For that reason most people begin by verifying their programs every time something is saved and end up never verifying anything because they have never had a problem. If you are naturally prone to caution then you will probably want to check what you have saved every time, if you are casual in your approach you probably never VERIFY your programs. The optimum course lies between the two attitudes.

When entering long programs VERIFY can take up a considerable amount of time. Losing the program can take up more and it does happen. Sometimes a program can be lost because of some limitation of the 64's SAVE routine, more often there will be some difficulty with the quality of the tape you are using. Such problems do not occur often but when they do they can be heartbreaking.

The best course is one of compromise. If you are confident of the quality of your tape then develop a program on something like a C60 cassette, recording each new version after the last version until you reach the end of the tape, then either turn over or rewind. In this way you would have to both lose the program on the machine *and* have a faulty recording in order to lose more than the last development of the program. When it comes to saving something like the final version, however, always VERIFY the tape.

4) While longer tapes are eminently suitable for developing problems there is no doubt that the best way to store programs permanently on tape

is to use specialist computer cassettes and limit each cassette to one program. This does cost a little more but it also means that all your programs will be instantly available, without having to search through long tapes for the right starting position. It also reduces the danger that you will unintentionally record over an existing program.

5) Keep the recording and playback heads on your cassette recorder clean. Kits for cleaning the heads are inexpensive and easy to use compared to the frustration of losing a program because the heads become coated with a deposit of oxide from your tapes.

6) Keep more than one copy of your programs, with the second copy in an entirely different place from the tapes or disks you normally work with. There is always the danger that your working copy will be damaged in some way, perhaps through excessive heat or the efforts of a child with a magnet. If you do not wish to duplicate every program on a single tape, your backup copies can be stored on a few longer cassettes and then rerecorded onto shorter cassettes if they are ever needed.

For disk owners the need to keep backup copies is accentuated. With the best will in the world the 1541 disk drive cannot be described as the most reliable that Commodore have ever made. It is not uncommon to suffer loading or saving difficulties or to have a disk accidentally damaged. Conversely the taking of copies is so much easier since programs can be saved to two disks with very little extra effort. Even if you do own a disk drive don't neglect the relative safety and reliability of tape for backup copies of important material. A serious disk drive fault can be extremely frustrating if your only copies of the required program are on disk.

Saving and loading data

Most programs which are of some actual use need a quantity of data to work upon. In cases where that data is not fixed so that it can be written into the program itself you have two choices, either re-enter the data every time or conquer the problems of saving data on tape or disk and reloading it into the program at a later date.

In deciding to save data to tape or disk the first problem to be encountered is that of identifying what it is you want to save. It is no use saving the contents of an array if you forget to save the variable which perhaps was associated with the array and recorded how many items there were in it. First of all, then, make a complete list of the essential variables in your program. This will not be all the variables in the program, it should be remembered. Many variables will be assigned values as the program runs. You only need to save those which are necessary to get the program back on the road when you next wish to use

it. One point to remember in this respect is to save only those parts of arrays which actually contribute something. It may be that you have defined a 500 line string array in which you are gradually building up a store of data. If you have only used 170 lines of the array so far it is best to ensure that you do have a variable recording of how many lines are used and save only those lines. The reason for this is that loading and saving data, whether to tape or disk, is not as fast, byte for byte, as loading and saving a program and there is no need to make this situation even worse by wasting time on non-useful items.

Saving to tape

Having identified the variables and parts of arrays you must set them out in a module which will reliably save them to tape. The first step here is to open a file with a line such as:

```
1710 OPEN 1,1,1,'FILENAME'
```

In the context of a micro like the 64 a 'file' is not a static location into which information will be placed but a line of communication. The three figures in the example line indicate that the file which is being opened will be referred to whenever something is to be saved as file 1 or '#1' for short, that it is meant to be a line of communication to 'device number 1', and that the type of communication to be opened up is where the 64 talks to the external device rather than the other way around. Other files can opened while this one is working, and you can have up to 10 files open at the same time though there are very few cases where more than one or two files are needed simultaneously.

When opening a file for storage there are two normal choices for a secondary address, namely one and two. A secondary address of one specifies that the file is an output file which will accept data as it is presented. A secondary address of two signifies that the file is an output file but with the added feature that a special 'end of file' marker will be added to the end of the data. This second form of the data file allows you to read items from a file without knowing exactly how many items there are, with a test line detecting the end of the file and ensuring that you do not generate an error by trying to input beyond the end of the data:

```
1710 NN=0 : OPEN 1,1,0,'FILENAME'
1720 INPUT#1,T
1730 IF ST=64 THEN CLOSE 1 : RETURN
1740 A(NN)=T
1750 NN=NN+1 : GOTO 1720
```

This routine will go on accepting items from the tape until the end of file marker is encountered. This marker will change the value of the system variable ST (STATUS) and the program execution will move on. A word of caution, however, using two as a secondary address can produce some strange effects on material stored after the file in question on the same tape. In most cases the EOF marker is not necessary, any decent program should be capable of recording how many items of data it is currently storing and therefore how many have to be stored. If this is known, variables representing the quantities of data can be stored at the beginning of each block of data and the program designed to load back that specific number of items, without the use of an EOF marker. The moral is: don't use a secondary address of two unless you really need to.

The general format for opening a file is then:

```
OPEN file number, device number, type of
file (known as 'secondary address'),'FIL
ENAME'
```

and any attempt to place information into a file which has not been opened in such a way will result in an error message.

It is not necessary to spell out the name of the file to be opened in the program line itself, it can also be accepted from the user in the form of a response to an INPUT statement:

```
1710 INPUT 'NAME OF FILE:';FF$
1720 OPEN 1,1,1,FF$
```

This allows the same program to create data files with distinctly different names. The technique can also be employed with the input of data to specify different input files and thus allow the program to shuffle between different files for different purposes.

Printing to a file

Having opened the file it is now necessary to start placing information into it. This is done using the PRINT# statement. Anything which is preceded by PRINT# and the appropriate file number will be placed into the file in much the same way that PRINT would place it on the screen (you can even print to the screen by opening a file to device 3) but there are a number of differences to be noted:

1) The 64, unlike many other micros, will not automatically mark the dividing line between variables which are PRINTed by the same line using

a statement like:

```
PRINT#1, A$,B$,C$
```

Such a line would result, when the data was recalled from tape, in a single string being detected, consisting of A$,B$, and C$ run together. In order to separate items which are to be saved there are two alternatives, either to print each item of data with a separate PRINT# statement or to insert a separator character between the items.

In the case of arrays, the separation of items is accomplished by using a loop to save each of the elements in turn, eg:

```
1710 FOR I=0 TO ITEMS
1720 PRINT#1,A$(I)
1730 NEXT I
```

Single items need to be printed to the file followed by a 'return' character (CHR$(13)). This is usually done by defining a string, say R$ as equal to CHR$(13) when the program is first initialised and separating every item with R$ to save typing 'CHR$(13)' every time:

```
1740 PRINT#1,TT$ R$ CD$ R$ IT R$ NN R$ Q
     Q$
```

Note here the lack of punctuation between the items. You can if you wish put commas or semi-colons between the items but the 64 is indifferent to them.

2) The 64 is not capable of saving and reloading characters outside the range of normal printing characters. If you wish to save strings which contain control characters of other than characters like cursor and colour controls which are a normal part of strings, you must translate at least a part of the string, character by character, into numbers and then store those numbers one at a time:

```
1710 PRINT#1,LEN(A$)
1720 FOR I=1 TO LEN(A$)
1730 PRINT#1, ASC(MID$(A$,I,1))
1740 NEXT I
```

Note that you have to store the length of the string in order that when reloading, the program will know when to stop expecting translated string characters.

4) You cannot save empty strings. This can cause some problems in the case of string arrays, which may often contain empty elements. What will happen if the first element of any array is empty is that nothing will be saved and the second element will effectively become the first on the tape thus disrupting the order of the array when it is reloaded. In order to overcome this the simplest solution is to 'pad out' every element in the array with a leading character before it is saved:

```
1710 FOR I=0 TO ITEMS
1720 T$='*'+A$(I) : PRINT#1,T$
1730 NEXT
```

Clearly you will have to remember to strip off these characters when reloading but padding every element is just as quick as testing every element to see whether it is a null string and then padding out only the null strings.

5) The order in which you save the data can be vital in determining whether it is possible to successfully reload it. Harking back to an earlier example, suppose that you have a string array of 500 elements and a variable to record how many of the elements are currently being used called 'ITEMS'. When you save the contents of the array you will use a loop such as:

```
1710 FOR I=0 TO ITEMS
```

When reloading, the program will therefore have to have a value for ITEMS before it can take the data off the tape again. The simple rule is that if any variables from the program are used to control the way the data is saved onto tape, then those variables should be stored *before* the date itself.

6) When all the data has been stored the file must be 'closed'. Failure to do this will mean that any subsequent attempt to open a file of the same number will result in the program stopping with an error message. The format for closing a file is:

```
CLOSE<file number>
```

Before closing a data file, however, it is wise to ensure that nothing is left in the 'memory buffer' which stores information as it is printed to the tape, otherwise the last item(s) of information may not be properly saved. This is meant to be accomplished by CLOSE alone but experience

shows that it is more reliably done by adding an extra command PRINT#<file number>, with no actual data specified. Thus the format for closing a data file becomes:

```
PRINT#<file number> : CLOSE<file number>
```

Loading from tape

The procedure for loading from tape is in many ways simply a mirror image of that for saving. The main differences are:

1) The type of file which is opened is different eg:

```
OPEN 1,1,0,'FILENAME'
```

where the zero indicates that this is a file which will be used by the 64 to receive information from an external device.

2) The main statements involved in accepting information back from tape are INPUT# and GET#. They will be explained more fully later.

3) Provided that proper separation has been made between the data items when they were stored there is no need to do anything about detecting the separation between items of data:

```
1840 INPUT#1,TT$,CD$,IT,NN,QQ$
```

would be sufficient to recover from tape the data stored in the example given above of the use of R$ to separate items.

4) If leading characters have been added to strings, they must be stripped off again:

```
1810 FOR I=0 TO ITEMS
1820 INPUT#1,T$ : A$(I)=MID$(T$,2)
1830 NEXT I
```

5) Strings which have been stored in the form of the numeric values of their characters must be reconstituted as strings:

```
1810 A$='' : INPUT#1,LS
1820 FOR I=1 TO LS
1830 INPUT#1,T : A$=A$+CHR$(T)
1840 NEXT I
```

6) Provision must sometimes be made for the limitations of INPUT# which, like the normal INPUT, can only accept string inputs of up to 88 characters. This is done by the use of GET#, which picks up the contents of the tape one character at a time, to recreate the original string character by character.

```
1810 A$=''
1820 GET#1,T$ : IF T$<>CHR$(13) THEN A$=
A$+T$ : GOTO 1820
```

In this case GET# will continue to pick up characters and add them to A$ until the end of string marker is detected.

7) In general the best place for the program module which reloads data is right next to the one which saves it. This is because the safest way to write a loading routine which exactly mirrors the order in which data was stored is to edit the line numbers of the saving routine and change PRINT# statements to INPUT#. Even so the two routines can be called by the program in many different ways.

The save routine should be a normal program function, called from the program menu, with perhaps a reminder to the user before the program is stopped (which again should be a menu option rather than simply pressing 'STOP') that is a good idea to save data before it is lost. If considerable bodies of data are being entered into the program the user will therefore be able to save the data regularly against the possibility of problems with the 64 or the program.

The load routine can be called in a very different way by the use of an 'auto-load' function at the very beginning of the program, similar in some ways to the 'auto-initialise' technique described in Chapter 1. In the case of auto-initialise the user has the two options of either RUNning the program, which means that the auto-initialise function will reset the variables to start from scratch, or of starting with GOTO, which will lead to the initialising module not being executed if there is already data held by the program. The auto-load function adds to these options by allowing the user, if there is no data held by the program, to specify whether new data is to be added from the keyboard or existing data first loaded from tape. If the user chooses to load data from tape then some parts of the initialisation routine will be carried out, namely the parts which set up the arrays to receive the data, but variables will not be set to their initial value since they will be loaded from tape anyway. Given below is an example of such a module:

```
1000 IF A$(0) THEN 1500
1010 DIM A$(100),B$(25),A%(100),B%(25) :
 LIMIT=100 : R$=CHR$(13)
```

```
1020 INPUT 'DO YOU WISH TO LOAD FROM TAP
E (Y/N):';Q$
1030 IF LEFT$(Q$,1)='Y' THEN GOSUB LOADE
R : GOTO 1500
1040 ITEMS=0 : NN=0 : CT=12 : BASE=2
```

Here the auto-initialise line is 1000. The second line dimension four arrays and declares a variable whose value will not be changed during the course of the program, in this case the maximum number of items permitted. Once these are declared the user can specify whether data is to be loaded from tape. Remember that if data is loaded from tape then that data must include all the variables which are passed over in the part of the initialisation module which is not executed. For this reason it is even more important than usual to spell out all the variables, even those with an intitialisation value of zero, in order to remind yourself when writing the save and load modules.

Save and Load routines: a working example

Given below is an example of a save/load routine taken from a program of my own. There is no need to try to understand the functions of the variables involved, the example is intended to illustrate only the method employed with complex bodies of data:

```
20000 REM*******************************
20010 REM DATA FILES
20020 REM*******************************
20025 R$=CHR$(13)
20030 INPUT 'POSITION TAPE CORRECTLY THE
N [RVS]RETURN';Q$
20040 IF NN$='' THEN 20140
20080 OPEN 1,1,1,'FILENAME' : PRINT#1,NN
$,R$,QQ$,R$,CU,R$,ITEMS
20090 IF CU=0 THEN 20110
20100 FOR I=0 TO CU-1 : PRINT#1,T$(I,0),
R$,T$(I,1),R$,T(I) : NEXT
20110 IF IT=0 THEN 20130
20120 FOR I=0 TO IT-1 : PRINT#1,A$(I,0),
R$,A$(I,1),R$,C(I) : NEXT
20130 PRINT#1 : CLOSE1 : RETURN
20140 OPEN 1,1,0,'FILENAME' : INPUT#1,NN
$,QQ$,CU,ITEMS
20150 IF CU=0 THEN 20170
20160 FOR I=0 TO CU-1 : INPUT#1,T$(I,0),
T$(I,1),T(I) : NEXT
```

```
20170 IF IT=0 THEN 20190
20180 FOR I=0 TO IT-1 : INPUT#1,A$(I,0),
A$(I,1),C(I) : NEXT
20190 CLOSE1 : RETURN
```

Here the user is given the opportunity to position the tape and then the module automatically detects whether data is to be saved or loaded according to whether the string NN$ contains anything — this is like auto-initialisation in that it is vital to choose a variable for this test which will always contain something when the program holds data. This 'auto-load' line could be replaced with a simple two-line menu allowing the user to specify whether loading or saving is intended. Two sets of arrays are then saved or loaded, with the amount of data to be handled based on the contents of two variables CU and IT. Finally the file is closed. Note that the format of the two halves of the module is exactly the same. The load routine was in fact copied from the save routine by changing line numbers and then editing the lines, thus helping to ensure that items would be recalled in exactly the order in which they were stored.

Variations for disk

Owners of disks will find the use of data files far more convenient simply because of the speed at which information can be loaded and saved. This is not only a question of the speed at which the disk operates, it is also because the disk drive will automatically position an output file on the disk or find the specified input file. The techniques for actually storing the data, separating items and so forth are the same but some extra provisions will have to be made for specifying the kind of file to be used and to overwrite the file if previous batches of data have been stored under the same filename.

1) As well as specifying the type of file with the numbers attached to the OPEN statement, the disk drive requires the filename to include an extra specification. In the case of data files the kind of file we are using for this type of application is known as a 'sequential' file, or one which simply accepts items of information in the order in which they are given and later can offer them back in the same order. To create such a file the format of the OPEN statement is:

```
OPEN 1,8,2,'FILENAME,S,W'
```

where the 8 refers to the device number of the disk drive, the S specifies a sequential file and the W indicates that the file is a 'write file' or one to which information will be output. Note that a secondary address of two

is normally used with disk files and the drawbacks noted in relation to tape do not apply.

When reading back information from such a file the format of the OPEN statement is:

```
OPEN 1,8,0,'FILENAME,S,R'
```

where the R indicates that this is a 'read file' or one from which information will be taken back into the the 64. If the W/R suffix is omitted, the disk drive assumes that a read file is intended.

2) The other complication that arises from the use of a disk drive is actually a result of the intelligence of the drive in knowing exactly what files are contained on its present disk. When using tape, as in the save/load module given above, the C2N recorder is not capable of detecting what, if any, new information is being written over. The user determines exactly where information will be stored by positioning the tape.

In the case of the disk a check is provided to ensure that the user does not accidentally wipe out an existing file by trying to save another with the same name. This is a valuable protection but there are often occasions where the same data file will be recalled from disk, added to or amended and then resaved, many times in the course of the continued use of a program. To allow for this a special prefix can be added to the filename so that it will overwrite any existing file of the same name and type. The format for this is:

```
OPEN 1,8,2,'@0:FILENAME,S,W'
```

Of course the facility can be combined with the ability to define the filename and with a user choice of whether a file will be overwritten or not:

```
1710 INPUT "NAME FOR FILE:";FF$ : FF$=FF
$+"S,W"
1720 Q$="N" : INPUT "OVERWRITE EXISTING
FILE (Y/N):";Q$
1730 IF Q$="Y" THEN FF$= "@0:"+FF$
1740 OPEN 2,8,2,FF$
```

Conclusion

There is no doubt that, though the outlines of storing and retrieving data are a simple matter, getting the data file module of a complex program right is often a fiddly job. That should not be allowed to detract from the knowledge that programs which are going to store credible amounts of

information need external storage. Nor should we all be fooled by the computer age into scorning the speed with which a disk drive or even a cassette can load large quantities of information into the 64's memory. Disks and cassettes are reliable and even relatively fast ways of storing useful information. Using them, no matter what their limitations, is certainly preferable to working with programs of the kind which are limited either to one set of information which is built into the program or to the information which can be entered at one session at the keyboard. If a set of information is worth working with it is in all probability worth the effort involved in storing it safely for future use.

CHAPTER 7
Logical Conditions

Few techniques can provide a greater saving in the length of program lines, or indeed make a program look more elegant than the proper use of the internal logic of the 64 in conjunction with more normal BASIC programming. In this chapter we enter the sometimes obscure world of the logical condition IF, and the logical operators AND and OR. Many micro owners use them regularly in their programs without ever realising their potential.

The humble IF

Every BASIC programmer uses IF — it is one of the most essential tools of programming. Even so, the use of IF is not always as straightforward as it seems or, to put it another way, there are often more straightforward ways of achieving results with IF statements than micro owners sometimes seem to realise.

The essence of IF is that an action can be performed or not according to whether a condition set by the programmer is fulfilled. Thus in a line such as:

```
100 IF A>10 THEN K=K+1
```

only if A is greater than 10 will the addition in the second part of the line be carried out. This is true for any part of a line which falls after an IF statement. In this way, within the limits of the length of a single 80 character program line, a series of commands can be carried out or ignored, eg:

```
100 IF A>10 THEN K=K+1 : X=X-10 : Y=Y*10
0 : GOTO 200
```

Where too many commands follow from a single condition to be expressed on a single line, the simplest alternative is to use the opposite condition to jump around a section of the program:

```
100 IF A<=10 THEN GOTO 120
110 PRINT "THE VALUE OF A IS NOW OVER 10
" : K=K+1 : X=X-10 : Y=Y*100 : GOTO 200
```

Protecting against illegal values

One use of the power of IF to isolate the part of the line which follows it is to protect against possibly illegal values of variables which would cause the program to stop. In the following example a BAD SUBSCRIPT error will be generated in line 20:

```
10 DIM A(20)
15 SS=25
20 IF A(SS)> 10 THEN 50
```

This problem can be avoided by making the original IF in line 20 itself subject to an IF:

```
20 IF SS<=20 THEN IF A(SS)>10 THEN 50
```

The only limit to the number of IFs which can be 'cascaded' in this way is the line length.

Errors arising from IF

A common error found in programs is to forget this power of IF to default around part of a line. Thus the intention of the programmer might be to scan through an array, subtracting 10 from all values over 100, but the line entered is:

```
100 FOR I=0 TO 99 : IF A(I)>100 THEN A(I
)=A(I)-10 : NEXT I
```

This would only in fact make the adjustment if every element encountered were over 100. The first time a value less than 100 is encountered the NEXT I will be ignored and the loop terminated.

IF...THEN...ELSE

One deficiency of the IF statement on the 64 is the absence of a feature which has become an accepted part of the BASIC on many other home micros, the IF...THEN...ELSE statement. In this optional format the user is permitted to specify two actions after the IF, one to be carried out

if the condition specified is true, the other if it is false. Thus a line such as:

```
100 IF A>10 THEN K=K+1 ELSE K=K-1
```

would add one to K if A were more than 10 and subtract one if it were not — *NB this will not run on your 64.* The usefulness of this command is that it is very often the case that the programmer needs to specify either/or actions in a program and other ways must be found to achieve this on the 64. One of the simplest ways is to specify one action before the IF and one after. Thus:

```
100 IF A>10 THEN K=K+1 ELSE K=0
```

can be simulated by:

```
100 TT=K : K=0 : IF A>10 THEN K=TT+1
```

If the actions to be carried out would take too much room to be included on the same line, two lines can be used with opposite conditions:

```
100 IF A>10 THEN PRINT "MORE THAN TEN IT
EMS" : K=K+1 : GOTO 200
110 IF A<10 THEN PRINT "TEN ITEMS NOT YE
T ENTERED" : K=0 : GOTO 250
```

IF with >, <and =

You may notice in the examples given that combinations of >= and <= are used. In setting out conditions in a line there is always a choice to be made between simplicity and the ease with which a program may be read. If an action is meant to be carried out if, for instance, A is 10 or less, then the condition could read:

```
IF A <= 10 THEN....
```

or it could be, in most circumstances:

```
IF A<11 THEN....
```

The second form is shorter but when you read the program you may well be confused by the 11 when the important value is 10. The other point to remember is that if you are working with non-integer numbers, ie

numbers that may not be whole numbers, <11 does *not* mean the same as <=10, since <11 would allow through any value between 10 and 11.

IF with the operators AND and OR

The power of the IF statement is vastly increased by the addition to BASIC of the operators AND, OR and NOT. The first two of these allow the programmer to combine a number of separate IF statements into the same line. In the lines

```
100 IF A>10 THEN 120
110 GOTO 130
120 IF B<=100 THEN K=K+1
```

both of the conditions specified must be met, or the second part of line 120 will not be acted upon. Using AND the lines can be combined into one:

```
100 IF A>10 AND B<=100 THEN K=K+1
```

A different problem is presented by the lines:

```
100 IF A>10 THEN 130
110 IF B<=100 THEN 130
120 GOTO 140
130 K=K+1
```

where *either* of the conditions being met would suffice for line 130 to be acted upon. Here we must make use of OR:

```
100 IF A>10 OR B<=100 THEN K=K+1
```

Combining AND and OR

ANDs and ORs can be combined on the same line to specify complex sets of conditions provided that some attention is paid to the order in which the 64 will actually assess them. Enter these lines:

```
100 A=10
110 B=100
120 C=50
130 PRINT '[CLR]'
140 IF A=10 AND B=100 OR C=10 THEN PRINT
  'OK'
```

Running the program will result in OK being printed, so it is clear that the first two conditions connected by the AND were sufficient to carry out the action specified even though C is not equal to 10. Now change line 140 to read:

```
140 IF A=10 AND B=150 OR C=10 THEN PRINT
 'OK'
```

and you will find that line 140 will now not print OK. Change line 140 again to read:

```
140 IF A=10 AND B=150 OR C=50 THEN PRINT
 'OK'
```

and everything works again, but what does it mean? There are two possibilities:

1) The true condition referring to C has replaced the false condition referring to B, so that the line is being read IF A=10 AND (B=150 OR C=50).

2) The true condition referring to C replaces both of the two conditions ANDed together, so that the line is read IF (A=10 AND B=150) OR C=50.

The only way to discover which is right is to make another change so that line 140 becomes:

```
140 IF A=20 AND B=150 OR C=50 THEN PRINT
 'OK'
```

Now we can see that the condition after the OR has replaced the two ANDed conditions as if they were one. The conclusion is that any conditions with ANDs between them must all be fulfilled as if they were one condition. On the other hand any condition preceded by an OR stands on its own.

This strict order in which AND and OR are evaluated can sometimes lead to confusion, with lines which appear to be sensible failing to produce the expected result. The answer to the problem is to use brackets if you are unsure of the way in which a series of conditions hang together. Consider this line:

```
140 IF (A=10 OR B=100) AND (C=50 OR D=20
) THEN PRINT 'OK'
```

Here the brackets isolate the second and third conditions from the AND in the line, and force the line to evaluate the first two conditions and the last two before thinking about the AND. If this is all still as clear as mud, then try adding a new line to the little routine just used:

```
125 D=20
```

Change line 140 to the last version given above and, not surprisingly, the program will print OK when it is run, since every condition in the line is filled. Now alter line 140 again to read:

```
140 IF (A=10 OR B=200) AND (C=30 OR D=20
) THEN PRINT 'OK'
```

and you will find that the routine still works. Since A is equal to 10, the pair of conditions in the brackets is fulfilled; since D is equal to 20, the second pair of conditions is fulfilled. The AND is now surrounded by two true conditions and so the line can be acted upon.

Unpacking complex conditions

If you have real trouble with complex sets of conditions, and many people do, then take the line which is given and convert it into a series of 'TRUE' and 'FALSE' conditions, then simplify those conditions according to the following rules:

```
1) TRUE AND TRUE = TRUE
2) TRUE AND FALSE = FALSE
3) FALSE AND FALSE = FALSE
4) TRUE OR TRUE = TRUE
5) TRUE OR FALSE = TRUE
6) FALSE OR FALSE = FALSE
```

As an example, assume as in the program above that A=10, B =100, C=50 and D=20. In this case the conditions in the line:

```
IF A=10 AND B=200 OR C=50 THEN .......
```

translate as:

```
TRUE AND FALSE OR TRUE
```

and using the rules set out above this can be first simplified to:

```
FALSE OR TRUE
```

and finally to:

```
TRUE
```

```
IF (A=10 OR B=200) AND (C=30 OR D=10) TH
EN.....
```

becomes:

```
(TRUE OR FALSE) AND (FALSE OR FALSE)
```

```
TRUE AND FALSE
```

```
FALSE
```

Setting limits

Conditions are often used to set limited ranges for a variable within which an action will be set. The format here is:

1) If an action is to be carried out if a variable is in the range 11-20 or an action is not to be carried out if the variable is out of the range:

```
100 IF A=>11 AND A<=20 THEN...
```

2) If an action is *not* to be carried out if a variable is in the range 11-20 or to be carried out if the variable is out of the range:

```
100 IF A<11 OR A>20 THEN...
```

Sometimes the need is to specify two possible ranges. This can often occur when testing the input of a character to a program, with the permissible characters being either the numbers 0 to 9 or the letters A to Z. In this case we need to use two pairs of conditions:

```
100 IF (IN$>="0" AND IN$<="9") OR (IN$>=
"A" AND IN$<="Z") THEN GOTO 200
```

IF with NOT

Few people regard NOT as a regular part of their programming, and indeed there are few things that it can accomplish that cannot be done in

other ways. The function of NOT is to reverse the effect of a condition and it can be used to make certain lines more readable. Take the case of the simulation of IF...THEN...ELSE, where two consecutive lines used opposite conditions. This could probably have been expressed more clearly by:

```
100 IF A>10 THEN PRINT "MORE THAN TEN IT
EMS" : K=K+1 : GOTO 200
110 IF NOT A>=10 THEN PRINT "TEN ITEMS N
OT YET ENTERED" : K=0 : GOTO 250
```

Using logical conditions

In the chapter on string handling a simple line was given to convert a spurious negative value given by the FRE function into the correct value. The situation was that if the value produced by FRE was negative, it needed to have 65536 added to it. This could have been done by a line such as:

```
K=FRE(0) : IF K<0 THEN K=K+65536
```

In fact what we had was:

```
K=FRE(0) : K=K-65536*(K<0)
```

If we assume that the two statements mean exactly the same thing, it tells us some interesting things about the second form. Clearly what is happening in the second form is that the 'IF K<0' is being replaced in some way by (K<0). In addition, since we shall sometimes wish to add 65536 and sometimes nothing, the (K<0) in the second form must in some strange way sometimes represent the value zero and sometimes the value minus one, otherwise the '–65536' will make no sense. How does this state of affairs arise?

The value of a condition

In assessing a condition in a program the 64, like any other micro, works on the basis that something is either true or false. In order to record its conclusion about a condition the 64 examines the expression which comes after the IF and gives it one of two values, –1 if the expression is true, and 0 if it is false. Take, for example, a program with two variables X and Y, with X equal to 7 and Y equal to 5. Now consider the following and think of them as statements rather like 'John is taller than Bill':

```
a) X=Y
b) X>Y
c) Y<=X
d) Y>=X
```

Fairly obviously, statements a) and d) are false, while b) and c) are true.

In the context of an IF statement, a) and d) would be assigned a value of zero, b) and c) a value of minus one. Where a condition like those given above has been included the action specified by the IF statement is carried out if the condition has been assigned a value of minus one. In fact anything which can be assigned a value can be used after the IF statement. For instance the line:

```
100 IF TT THEN GOTO 120
```

would result in a jump to line 120 if the value of TT were not only minus one but anything other than zero. Strings too can be treated in this way:

```
100 IF A$ THEN GOTO 120
```

would be acted upon if A$ were anything but an empty string. This technique can be used in a variety of ways where it is important to detect a zero or null value. One application which is commonly used in error checking within a program and this is described in the chapter on error-checking techniques.

Using conditions as values

In addition to being able to use a single variable to activate an IF statement, another interesting area is opened up by the way in which conditions are evaluated as zero or minus one. In fact, conditions can be used anywhere in a program to give one of the two values, not simply in an IF statement. The FRE converting line above relies on this fact. Whenever a condition is encountered during the execution of a program it will always be assigned a value and this value can be used in controlling the program in just the same way as any other variable. In the case of the FRE function above, 65536 is added to the result of FRE(0) only when the condition (K<0) is true. The beauty of such conditions is that they can be used in combination with each other to replace a mass of IF statements. Take the following lines:

```
100 K=100
110 IF TT>120 THEN K=K+50
120 IF XX<50 THEN K=K-25
```

```
130 IF Y$='DEBIT' THEN K=K+100
140 IF FF=0 THEN K=K*ZZ
```

These, and many more of the same could be replaced by lines such as:

```
100 K=100-50(TT>120) + 25*(XX<50) - 100*
(Y$="DEBIT")) * (ZZ+(ZZ-1)*(FF<>0))
```

Plus or minus?

One confusing thing here is that the plus and minus signs appear to be the wrong way round. If you think about it, this has to be done to take account of the fact that the result of a true condition is not one but minus one. If you want to add 100 to K if something is true, you have to subtract 100 times its value when true (ie minus one), so that the result is K–(–100), or K+100.

Multiplying and dividing

Note also that when multiplying (or dividing) according to the result of a condition, a straightforward method is to take the number you wish to multiply by and first subtract from it one less than its own value times the value of the condition opposite to the condition chosen. Thus if you wanted to multiply by 1000 if X was equal to zero, then you would in fact multiply by 1000+999*(X<>0). If X *was* equal to zero then you would be multiplying by 1000+0, or 1000. I X *was not* equal to zero you would be multiplying by 1000+(–999), or one.

Avoiding isolation by IF

A point to remember is that conditions can be used to overcome the fact that IF isolates everything that follows it when the IF condition is not fulfilled. Earlier on we noted the danger of using IF in the same line as the NEXT which terminates a loop if the intention is for the whole of the loop to be executed. Such problems can, however, be overcome by a line such as:

```
100 FOR I=0 TO 99 : A(I)=A(I)+10*(A(I)>1
00) : NEXT
```

There are some cases, it should be remembered, where it is not appropriate to replace IF statements with expressions based on the value of conditions. Where two IF statements are used to protect a variable from being accessed if it has an illegal value, the use of conditions will

not provide that protection. In a previous section we saw that a line such as:

```
100  IF SS<=20 THEN IF A(SS)>10 THEN K=K
+1
```

protects the program from crashing if the value of SS is greater than the maximum number of elements in the array. A line such as:

```
100  K=K + (SS <= 20) * (A(SS)>10)
```

would allow the program to access the illegal value of SS and thus crash. Other than such cases of protecting a program such successive IFs can anyway be replaced by the use of AND. Such lines *can* be replaced:

```
100 IF A=10 AND B=20 AND C=30 AND D=40 T
HEN K=K+10
```

could be replaced by:

```
100 K=K+10 * (A=10) * (B=20) * (C=30) *
(D=40)
```

Two things are worth noting here, firstly that the savings in program space are much less extensive and secondly that the rule with regard to the sign of the number to be added is apparently broken. This is because we have to take into account the number of conditions to be multiplied together. The rule here is that if a number of conditions to be multiplied together is *even,* then the resulting number is added, if it is *odd* then the resulting number is subtracted.

Lines relying on conditions are not always easy to read but they are much easier than they look to write. Such difficulties, however, are outweighed in many circumstances by the compactness of the programs which can be written with their aid.

AND and OR with numbers

One added and very useful effect of AND and OR is in relation to numbers, where they can be used to produce some arithmetical results that would be extremely cumbersome by any other means. Used in this way AND and OR perform their particular magic on the individual bits of binary numbers up to 15 bits in length (0–32767). When two numbers are ANDed, the result is a number which is made up of the binary digits which were set to one in both of the original numbers. When two numbers

are ORed the result is a number with every binary digit set to one which was set in *either* of the original numbers. Thus:

```
231 (Binary 11100111) AND 126 (Binary 01
111110)=102 (Binary 01100110)
```

and

```
231 OR 126 = 255 (Binary 11111111)
```

This seemingly obscure ability can be used in a variety of ways within a program.

POKE with AND and OR

Many functions on the 64, like sound and sprites, can only be accessed by use of POKE statements. Very often the object of such POKEs is to change a single bit within one byte of memory, without affecting the other bits. This can be done easily by use of AND and OR. The rule here is that to turn on one bit in the byte at location ADD, a line such as POKE ADD, PEEK(ADD) OR 2^BIT must be used. If the need was to ensure that bit zero (the bit that represents 1 in a number) at location 53287 was switched to one, or 'set', then the line would be:

```
100 POKE 53287, PEEK (53287) OR 2^0
```

The 2^0 could more easily be written as 1 but using the format as shown ensures that you do not mistake the bit to be altered.

To reset or switch to zero a bit requires the use of AND, and the rule here is that the contents of the location in question are ANDed with 255–2^BIT, as in:

```
100 POKE 53287, PEEK (53287) AND (255-2^
0)
```

Storing with AND/OR

Using AND and OR a single variable can be very effectively used to store up to 15 'yes/no' items of data. If the variable is first declared as equal to zero, any one of the 15 bits can be set by a similar technique to that used when poking. To set bit 7, the following line would be used:

```
100 A=A OR 2^7
```

and to reset the same bit:

```
100 A=A AND (255-2^7)
```

The variable can then be read by the use of a simple loop:

```
100 FOR I=0 TO 14:IF A AND 2^I THEN K=I
: GOSUB 1000
110 NEXT I
```

A single bit can be tested by a statement such as:

```
100 IF A AND 2^X THEN.....
```

In one of my own programs which stores payments and receipts from a bank account, a single two-byte variable for each payment records which month of the year it was made in.

Odd or even?

A common requirement in programs is to test whether a variable is odd or even. This too is a simple matter of testing a bit, in this case bit zero of the variable, as in:

```
100 IF A AND 2^0 THEN...
```

Conclusion

This is a bitty chapter, full of snippets which seem to lead in many different directions. Nevertheless, as you look through your own programs I hope you will find many opportunities to replace bulky and inefficient coding with shorter, clearer and more elegant instructions. If nothing else, you will be less puzzled when you set out to understand some of the programs laid out in magazines and books, for if the programmers know what they are doing then the kind of material laid out here will always play an important role.

CHAPTER 8
A Jungle of Sorts

This chapter is about where I came into writing for micro computer owners. A couple of years ago a national computing magazine in Britain published a program designed to allow users to store the names and addresses of their friends. When all the names were entered, at the push of a button the program would sort them into alphabetical order for future use. I suppose the program was published because it was neat and fairly short but the moment I saw it I had my doubts about the sorting method employed.

Rather than enter the maximum 100 names allowed, I sat down and wrote a simple routine which generated 100 nonsense names and addresses. With those completed I ran the program. *Half an hour* later it finished sorting. If I had entered only 90 names and come back later to add one or two, it would have taken nearly the full half hour just to get those extra names into place. A useful program ruined for all practical purposes by the use of an inadequate method.

Using a sheaf of notes from a college course in computing, I wrote a short article showing how the program in question could have been speeded up by a factor of around 40, the first full article I ever had published. Since that time I have never ceased to be amazed at the number of applications programs written by micro owners which are marred because the writers are unaware of the variety of sorting methods available for different purposes and the corresponding variation in the amount of time that they take. Admittedly that magazine program was written for one of the slowest machines ever to be put on the home micro market. With a machine as sophisticated as the Commodore 64 you would have to work very hard to slow down a sort to the extent that it would take half an hour for only 100 items. The principle, however, still applies. An appropriate sorting method can make the difference between a program which is painfully slow and one which is fast enough to be of some use.

In this chapter we shall examine three methods of sorting showing how and why they work, and why it is that the time they take can differ quite so much. Before we look at sorting on the computer, however, we shall take a more basic look at what happens when a human being sorts items,

hoping to gain a better understanding of what we shall be looking for when we turn to the keyboard.

The whys and wherefores of sorting

To begin our simple experiment you will need to find yourself 10 pieces of paper, filing cards would be ideal but if you don't have any then simply cut up a sheet of paper to give yourself 10 roughly 3 inch squares. Onto your 10 slips write the numbers 0 to 9, remembering to put a line under the 6 and 9 so that you know which is which if you inadvertently turn them upside-down. Now find yourself a clear area of table (preferably away from any draughts) and lay the slips out in a line in the following order:

7 3 5 1 6 9 4 8 0 2

(There is nothing special about the order laid down but if you don't use the same sequence then the comments below will not make much sense.)

The object of the exercise is to sort the slips into ascending order, 0 to 9, beginning at the left. The only limitations are that you have no more than 11 spaces to play with, the 10 occupied by the slips on the table and one spare space. That means that before you can move any slip from one place to another in the order you must first remove a slip from the sequence and place it in the eleventh position. That will give you one space in the sequence to which you can move a slip and one slip entirely removed from the sequence in the spare space. At the end of the process your aim is to have all the slips in order and the spare space empty. Try it and, as you do, attempt to analyse what it is that you are doing.

If you have actually followed the procedure laid down so far then what you have probably done is:

1) Scanned the sequence to find either the highest or the lowest value.

2) Removed the slip in the highest or lowest position to the spare place and moved the correct slip into the now empty place in the sequence.

3) You now had a choice between placing the spare place slip into the gap in the sequence and starting again from 1) for the second highest or lowest or identifying the correct place for the slip in the spare position and moving the present slip from that position and correctly placing the spare slip, thus saving on the eventual number of swaps to be made.

Perhaps you worked very differently but if you made the best use of your brain and eyes a number of things will be true about the way in which you made the sort:

1) Because the slip would eventually occupy a position in the sequence corresponding to their numbers you were always able to see quickly where any particular slip should go.

2) Because there was a relatively small number of slips you could easily identify the eventual highest and lowest in the sequence and could act accordingly.

3) You were capable of seeing the whole of the sequence, almost at a single glance, so you could shape your actions accordingly.

Now try to place yourself in the position of a micro computer beginning on the process of a sort, the kind of position that you might find yourself in if you were faced with 100 filing cards, each bearing say, the name of a different person:

1) You could count on no regular order which would enable you to identify instantly where any particular card might be placed. There might in fact be such an order but you would have no way of knowing it. Of course it would be possible to write a very fast sort on the assumption that if there were 100 items then their values would be spaced absolutely regularly, so that looking at a card would instantly tell what its eventual position would be. The problem with such a sort is that it would only work with such an orderly list. A normal sorting must deal with every kind of data and, unlike a human brain, it cannot say 'ah, an orderly sequence of numbers which will allow me to place each item correctly the first time'.

2) Because there was no particular regularity to the data, you would be unable quite so easily to identify the highest or lowest item in the eventual sequence, then the second highest and so on. If you wanted to find the highest value card, you would have to examine every card and you would only be able to decide at the end which had been the highest.

3) Because you can only carry out one task at a time, examine one fact at a time, you would never be able to get a picture of the whole list. You would have to compare an item here and an item there, without having a picture of what was going on around them. Give a human being a list in the order:

0 1 2 3 4 5 6 7 8 9

together with the instruction to arrange it in order 0 to 9 and back will come the instantaneous answer 'it already is'. The micro could never give that answer without first examining the list item by item.

In the light of these major differences between the ability of a human being and that of a computer we can almost classify sorting methods on a scale which has at one end methods which accept the limitations of the computer completely and at the other end, those which try to imitate some of the short cuts which the human mind would take. The simplest of all the common methods, and the most crudely 'computer-like' in its operation, is the Bubble Sort.

The Bubble Sort

The essence of the Bubble Sort is a complete reliance on the ability of the micro to compare two adjacent items and decide which of them should be higher. It takes its name from the way in which, as the sort proceeds, higher values seem to 'bubble up' the list in much the same way as the bubbles on the wall of a glass containing an effervescent drink.

For a demonstration of exactly what is meant by this we return to our slips of paper, which should be laid out again in the order:

7 3 1 5 6 9 4 8 0 2

Observing the same limitations as before, namely that there is one spare space available for you to use and that it must be empty at the end, follow this procedure:

1) Starting with the first slip, the 7 on the extreme left, compare it with the slip in second place, in this case a 3. Since the 7 is bigger than the 3, remove the 3 and place it in the spare position. Move the 7 to the position originally occupied by the 3 and then move the 3 back from the spare place to the position originally occupied by the 7. You have now moved the 7 one place up the sequence.

2) Follow the 7 up the sequence, performing the same procedure every time you find a smaller number to the right of it. Eventually you will end up with the 7 in the fifth position in the sequence and the 9 to the right of it.

3) Because you have found a number bigger than the 7, ie the 9, you transfer your attentions to that new number and treat it as you did the 7, swapping it with the number to the right if that number is smaller. Because 9 is the largest number in the sequence, you will carry on until the 9 is placed in the highest position.

4) Go back to the beginning of the list and start again with the 3. If the number to the right of it is smaller, then swap them, if not forget about

the 3 and continue with the larger number. At the end of the process you should have the 8 in position 9, to the left of the 9. Because you have already discovered the highest number on the scan through the list, there is no actual need to compare with the final number on the list.

5) Go back to the beginning and start with the 1. You will immediately have to change it for the 3. Carry on up the sequence, remembering that this time you can leave the last 2 items out of account, since they were correctly placed by the previous scans.

6) Repeat the procedure until you go through the whole list once without making any swaps. Remember that, as a computer, you do not know that the list is in order until you have done this — you cannot see the whole list.

If you have followed the procedure correctly, the slips should look like this at the end of each of the scans through them:

```
7 3 1 5 6 9 4 8 0 2  :  START POSITION
3 1 5 6 7 4 8 0 2 9
1 3 5 6 4 7 0 2 8 9
1 3 5 4 6 0 2 7 8 9
1 3 4 5 0 2 6 7 8 9
1 3 4 0 2 5 6 7 8 9
1 3 0 2 4 5 6 7 8 9
1 0 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Once you have run through the procedure and finished up with a correctly ordered sequence, you might like to try some other random sequences of your own to familiarise yourself with the method. There is, however, something more to learn which is of vital importance in assessing this or any other kind of sort.

Set up the original sequence again and begin to sort it using the Bubble Sort method, but this time note down on a piece of paper, in separate columns, each time you make a comparison between two items (regardless of how the comparison comes out in terms of relative size) and each time you actually swap two of the slips. By my counting, the result is as follows for the 8 scans on which something is changed plus the one extra which tells you that the list is now in order:

```
1) 9 COMPARISONS   8 SWAPS
2) 8    '    '     4   '
3) 7    '    '     3   '
```

```
4) 6     '     '          3     '
5) 5     '     '          2     '
6) 4     '     '          2     '
7) 3     '     '          2     '
8) 2     '     '          1     '
9) 1     '     '          0     '
TOTAL:   45 COMPARISONS   25 SWAPS
```

You have now learned something crucial to any understanding of the operation of all the many sorts. Sorting requires comparisons and exchanges and the differences between sorts are entirely comprised of the number of these, with exchanges consuming far more time in their execution than comparisons. It is possible to assess sorts mathematically according to their likely performance and, while we shall not attempt to actually do that here, the results for the Bubble Sort are that in the best case (ie where it is given a list to sort that is already in order), to sort a list of N items it would require 0 swaps and N–1 comparisons. In the worst case (one where the list of items was in entirely the reverse order) the Bubble Sort would require 0.5*N*(N–1) swaps and the same number of comparisons.

Unpacking that little formula means that, in the worst case, the Bubble Sort would require:

For 100 items: 4950 swaps, 4950 comparisons.
For 1000 items: 499,500 swaps, 499,500 comparisons.

From that you can see that as the length of the list of items to be sorted increases, the Bubble Sort starts to become horrendously expensive in terms of the number of actions that have to be accomplished.

Of course, the actual number of swaps which have to be made in sorting a particular list will very seldom represent the best case or the worst case. We saw in sorting our own little sequence that we had the maximum number of comparisons [0.5*10*(10–1)=45] and this will often be the case with a list. In terms of swaps, however, we had only just over half the theoretical maximum. Lists will differ but in general the larger the list, the worse job the Bubble Sort will make of getting it ordered quickly. For small lists any difference in time may hardly be noticeable and may be outweighed by the simplicity with which the sort may be programmed.

Programming the Bubble Sort

Having taken the time to get a grasp of what sorts are about and in

particular how the Bubble Sort works, it is now possible to get down to showing how it can be expressed in a program (should you care to after what has been said). Given below is the skeleton of a short program which we shall use to test 3 different types of sorts. The program consists of:

1) A random word generator set to provide a disorderly list of 100 nonsense letter combinations, stored in the A$.

2) A copy routine which will copy the original list into a second array, B$, so that we can eventually use the same list for all of the sorts and compare their performance.

3) An order checking routine which tests to see that the list, as processed by one of the sorts is in the right order.

4) A routine to print out the list so that you can check its order visibly. You may not wish to actually call this routine unless you run into a problem, since it will prevent the timings of all the three sorts which will eventually be entered being displayed on the screen at the same time.

5) A routine which will perform a Bubble Sort on the disorderly list and place it in alphabetical order.

Note that in this chapter we shall always be sorting strings. To sort numbers, all that is required is to change the names of the arrays to numerical arrays. You will find, if you do that, that the sorts will be faster, expecially those which require a great deal of swapping, since constant swapping of strings on the 64 results in pauses to tidy up the memory at fairly regular intervals. There are some sorts which, on the 64 at least, are fast for numbers but impractical for strings due to the amount of swapping which takes place.

In the sorts given we also shall always be working to order our lists in alphabetical order. If you wish to order a list in the other direction (ie beginning at Z and working down to A) then all that has to be done is to change the lines which contain > and < conditions so that the conditions are reversed.

```
1000 REM**********************************
1001 REM CONTROL ROUTINE
1002 REM**********************************
1010 GOSUB 5000
1020 GOSUB 4000:TI$='000000':GOSUB 8000:
PRINT TI$:GOSUB 6000:GOSUB 7000
1100 PRINT 'LIST NOW SORTED'
```

```
1999 STOP
4000 REM*********************************
4001 REM COPY LIST
4002 REM*********************************
4010 FOR I=0 TO 99
4020 B$(I)=A$(I)
4030 NEXT I
4040 RETURN
5000 REM*********************************
5001 REM GENERATE LIST
5002 REM*********************************
5010 IT=100 : DIM A$(IT-1),B$(IT-1)
5020 FOR I=0 TO IT-1
5030 T$=''
5040 FOR J=1 TO 4+INT(9*RND(0))
5050 T$=T$+CHR$(65+26*RND(0))
5060 NEXT J
5070 A$(I)=T$
5080 NEXT I
5090 RETURN
6000 REM*********************************
6001 REM PRINT LIST
6002 REM*********************************
6010 FOR I=0 TO IT-1
6020 PRINT B$(I)
6030 IF I/10 = INT(I/10) THEN GET T$ : I
F T$='' THEN 6030
6040 NEXT I
6050 RETURN
7000 REM*********************************
7001 REM CHECK ORDER
7002 REM*********************************
7010 FOR I=0 TO IT-2
7020 IF B$(I)>B$(I+1) THEN PRINT 'OUT OF
 ORDER AT';I
7030 NEXT I
7040 RETURN
8000 REM*********************************
8001 REM BUBBLE SORT
8002 REM*********************************
8010 FOR I=IT-1 TO 1 STEP -1
8020 FOR J=0 TO I-1
8030 IF B$(J) <= B$(J+1) THEN 8070
```

```
8040 T$=B$(J)
8050 B$(J)=B$(J+1)
8060 B$(J+1)=T$
8070 NEXT J
8080 NEXT I
8090 RETURN
```

Provided that you have followed through the steps suggested above, these lines should present you with no problems, for they follow exactly the method which you have already worked through by hand.

The sort routine as listed can easily be lifted out of the program and used in your own applications where small quantities of data need to be sorted. All that will be needed is to change the line numbers to conform with your own program and the names of the arrays to those you wish to work with.

To use the program simply RUN it and wait for it to confirm that the random list generated has been correctly sorted. You will be given a timing which was begun when the copy of the random list was sent to be sorted and ended when the sort was complete. You might try adapting the program to work on larger lists by changing the value of IT in line 5010, provided that you have no objections to twiddling your thumbs while the sort chugs along. If you do decide to work with lists which are substantially longer than 100 items, it might be wise to insert an extra line somewhere incorporating FRE (0) which will ensure that the 64's garbage collection is carried out regularly and is not caught out by the immense demands of juggling hundreds of different length strings to the extent that the program stops with an OUT OF MEMORY error.

The Delayed Replacement Sort: a simple short cut

If you can still remember as far back as the first time we used the slips to try and determine how a human being sorts a list, I commented that one of the methods that most people would probably use would be to identify the highest or lowest value in the sequence and put that in its correct place immediately. You could have chosen to begin by putting the third slip in its correct position, then the seventh, then the first, in fact you could have done it in any order you liked. That is because the list had values with a regular gap between them and you could tell where each slip should go simply by looking at its number. If the gap had been irregular then starting by finding the correct slip to go in the third position would have been more difficult. In such lists, and most lists do have items with irregular intervals between them, the easiest thing to do if you want to identify the position of one item without ambiguity is to first find the highest or the lowest, then the second highest or lowest and so on.

In effect, that is what the Bubble Sort did. On each scan, it picked up the highest value which had not yet been correctly placed and deposited it in its appropriate place. While doing it, the sort also made some changes in the order of the rest of the sequence but the majority of the swapping going on was all about moving one item to its correct position. The question arises, is all that swapping necessary. The answer is no.

Let's go back to the slips to examine how many of those swaps might have been eliminated. First lay the slips out in the same order as we used to test the Bubble Sort:

7 3 1 5 6 9 4 8 0 2

Now find a small coin and place it on the leftmost slip, the 7. The coin represents the position of the highest value slip that you have found. The procedure is as follows:

1) Compare the value of the slip with the coin on it with the value of the slip to the right of it. The next slip, the 3, has a lower value so the coin stays where it is.

2) Move the comparison on to the right, in our case to the 1. Go on moving the comparison as long as the slip you are comparing has a lower value than the slip with the coin on it.

3) When you reach a slip with a higher value than the one which currently has the coin on it, in our case the 9, move the coin along to that slip. Then start comparing the new highest slip with those to its right, leaving the coin where it is if their value is lower and shifting it if their value is higher. In our case the coin will remain on the 9 until the end of the scan along the slips.

4) Take the slip at which you ended the scan and place it into the spare position. Place the slip with the coin on it into the empty space and then take the slip from the spare position and place it in the space left by the slip which had the coin.

5) Place the coin on the leftmost slip and begin the whole process again with the exception that, as with the Bubble Sort, you can reduce the number of comparisons on each scan to take account of the fact that previous scans have reduced the disordered section of the sequence by one each time.

If you follow the procedure described above you should obtain the following sequences for each scan:

```
7 3 1 5 6 9 4 8 0 2  :  START POSITION
7 3 1 5 6 2 4 8 0 9
7 3 1 5 6 2 4 0 8 9
0 3 1 5 6 2 4 7 8 9
0 3 1 5 4 2 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 3 1 2 4 5 6 7 8 9
0 2 1 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

If you feel that you have grasped the method then the real lesson about this sort is waiting to be learned, for now we turn to the count of swaps and comparisons in the same way that we did for the Bubble Sort. In this case, to be fair to the Bubble Sort, we shall also count the number of times the coin has to be moved plus one for placing on the leftmost slip to begin with. The resulting figures, as I count them anyway, are as follows:

```
1) 9 COMPARISONS  1 SWAPS  2 MOVES
2) 8   '     '    1   '    2   '
3) 7   '     '    1   '    1   '
4) 6   '     '    1   '    4   '
5) 5   '     '    1   '    3   '
6) 4   '     '    0   '    3   '
7) 3   '     '    1   '    2   '
8) 2   '     '    1   '    2   '
9) 1   '     '    0   '    2   '
TOTALS:  45 COMPARISONS  7 SWAPS  21 MOVES
```

Clearly what we have done is to make a major trade-off between swaps and moves of the coin. Even with our manual simulation, to move the coin is far simpler than to swap two slips. When the method is computerised, the saving represented by simply changing the value of a variable (the one used to record the position of the highest value found so far) compared to the three-part operation involved in swapping two strings becomes overwhelming.

The worst case figures for the Delayed Replacement Sort make the position even clearer. For the worst possible kind of list (for this sort at least), the number of comparisons will be, as with the Bubble Sort, 0.5* N*(N–1). The greatest number of swaps possible, however, is only N–1, or nine in our case. The greatest number of moves is actually achieved when the list is in fairly good order. For instance one scan of a list which is in perfect order will yield nine moves of the pointer to the highest value. If we keep a record of the number of moves made in any one scan and

the number of comparisons to be made in the same scan then whenever the two values are the same we are assured that the lower portion of the sequence is in perfect order. In the case of our slips, arranging them in the order:

1 2 3 4 5 6 7 8 9 0

we shall find that there are 45 moves, which looks suspiciously like our old friend 0.5*N*(N–1) again. We can therefore conclude that the worst cost of a Delayed Replacement Sort will be:

For 100 items: 4950 comparisons, 99 swaps and 4950 moves.
For 1000 items: 499,500 comparisons, 999 swaps and 499,500 moves.

If it is true that moves are considerably faster than swaps then the loss of some 500,000 swaps for the larger figure must reduce in a noticeable saving in time. This can only be tested by comparing the two sorting methods in action and, to do that, all that is required is to enter the routine given below, which slots into the program already given:

```
1030 GOSUB 4000:TI$='000000':GOSUB 9000:
PRINT TI$:GOSUB 6000:GOSUB 7000
9000 REM*********************************
9001 REM DELAYED REPLACEMENT SORT
9002 REM*********************************
9010 FOR I=IT-1 TO 1 STEP -1
9020 NN=0
9030 FOR J= 1 TO I
9040 IF B$(J) > B$(NN) THEN NN=J
9050 NEXT J
9060 T$=B$(I)
9070 B$(I)=B$(NN)
9080 B$(NN)=T$
9090 NEXT I
9100 RETURN
```

Repeated running of the sort will demonstrate that, on a rough average, it will provide a saving of some 50-60 per cent over the straight Bubble Sort for 100 items. At the same time it is clear that all we have really done is tinker with the method, taking out a few redundant swaps.

The Shell-Metzner Sort: harnessing the power of two

To achieve real power in sorting we must leave behind the relatively safe world of repeated scans of the data in an orderly fashion, seeking highest and lowest values. As with the Binary Search, we must launch out into an apparently patternless method in the trust that binary methods will bring order out of what seems at first to be chaos. In this sorting method, apparently crazy swaps will be made in the initial stages and yet the correct order will emerge from the mess in far less time, for substantial lists, than by either of the other methods we have employed.

Understanding the progress of the sort is not going to be easy but we shall attempt to perform it on our disordered slips as before. This time, however, you will need 4 coins of different values and a piece of paper to supplement the slips. We shall call the four coins, in ascending order of value, A, B, C and D. On the piece of paper we shall record successive values of a variable which we shall call GAP.

The basic method of the sort is that we begin trying to make swaps between items which have a gap between them of the greatest power of 2 which will fit into the list of data to be sorted. When we have exhausted all those swaps we proceed to make all the swaps of half that gap and so on. In our case the initial gap across which we shall attempt to make swaps is 8, so write on your piece of paper 'GAP=8'. Now place coins A and B next to the leftmost slip (7). During the sort we shall use the A coin (the lowest in value) to indicate the leftmost of the two slips we are trying to swap and the B coin to record how far we have scanned along the array for each value of GAP. The C coin is now placed on the slip which represents the farthest to the right from which a swap of GAP can be made. In our case that will be position 2, since 2 plus the value of GAP takes us to the far end of our array of slips. The D coin will be moved about during the sort to show the position of the slip GAP places to the right of the slip with the A coin on it.

Now follow the procedure laid down below:

1) Lay out the slips in the order 7 3 1 5 6 9 4 8 0 2, as before.

2) A is in position 1 so D must be placed in position 1+GAP (=9). These two slips are 7 and 0 so they are swapped. The coins are not moved yet.

3) Silly as it may sound at this stage, we now examine whether it is possible to move A GAP places to the left. I shall explain why later. For the moment, simply note that we consider it, find it to be impossible and therefore move B one place to the right.

4) Whenever B is moved one place to the right, A is automatically moved to the same position, so place A in position 2 with B. This is also the position occupied by C.

5) D is now moved to A+GAP, which means 10. The slips are 3 and 2, so they are swapped.

6) Once again we consider the possibility of moving A GAP places to the left. It cannot be done so we move B one place to the right.

7) You will notice now that B has moved ahead of C. This means that the first scan of the data is complete. Whenever a scan is complete we must reduce the value of GAP by half and reset all the coins. Cross out 8 on your piece of paper and replace it with 4. Move B back to the leftmost position, followed by A. C must be placed in position 10-GAP, which is now 6, representing the rightmost position from which a swap of GAP 4 could be made.

8) Now we begin again trying to make swaps. Move D to A+GAP, which will be position 5. The slips are 0 and 6, so they cannot be swapped. Whenever a swap cannot be made B is moved one place to the right, with A following it to its new position. D is moved to position 6 (A+GAP).

9) The process of moving B, A, and D is repeated until B and A are in position 6. Up until that point it has been impossible to make any swaps of GAP 4.

10) With B and A in position 6, the two slips indicated by A and D have values of 9 and 3, so they can be swapped.

11) Since a swap has been made, we consider moving A GAP places to the left. This time it is possible, so we do it. A is now in position 2 so we move D to A+GAP, which is 6. The two slips are 2 and 3 and cannot be swapped. The reason we tried this is that whenever a slip is moved down the sequence, we need to test whether it could have been swapped if it had been in its new position from the start. If the swap had been possible then we would have tried to move A GAP places to the left again. While this is happening, B remains where it is.

12) Whenever a point is reached when a swap cannot be made or A cannot move GAP places to the left, B is moved, with A following it. This means that B is now ahead of C and the scan is therefore ended.

13) Reduce the value of GAP by half to 2. Replace A and B in position 1. Place C in position 10–GAP (8). Place D in position A+GAP (3).

14) Swaps can now be tried unsuccessfully until B and A reach position 4, with D at 6. Swap the 5 and the 3. A moves back to position 2 but

another swap cannot be made, so move B one place to the right.

15) With B and A now in position 5 you can swap the 6 and the 4. No swaps are possible when A is shifted to the left so B is moved.

16) No swaps are possible for positions 6,7, and 8 so the scan ends without further swaps being made.

17) GAP is reduced to 1, A and B are replaced in position 1, C is placed in position 10–GAP (9) and D placed in position A+GAP (2).

18) In effect, moving the pointers along at GAP 1 will produce one final scan of the type used by the Bubble Sort. You should be able to accomplish the coin moves for this without further guidance.

19) If all has gone well the list is now ordered, with the indication of this being that GAP would be less than 1 for the next scan.

20) As one final touch, to illustrate why the A pointer is moved to the left whenever a swap is made, you might like to try that final, Bubble Sort type scan on five of the slips arranged in the order 0 1 3 4 2. Use the coins as above and GAP set to 1. You will find that no swap is possible until B is in position 4 and that stopping here does not finally order the list. Moving A GAP places to the left (to position 3) accomplishes the extra swap which is needed to place 2 in its correct position.

We shall not attempt, for the Shell-Metzner Sort, to analyse the number of swaps, comparisons and moves, partially because it might take all day and partially because for this amount of items it would tell us very little, since the advantages of this sorting method do not become clearly apparent until larger bodies of data are sorted. That does not mean that the Shell-Metzner Sort cannot be used for small quantities of data, it is simply a matter of deciding whether the infinitesimal increase in speed is worth the extra programming involved.

When you come to enter the Shell-Metzner Sort into the sort testing program with the routine that follows, you will find that the savings are dramatic for larger bodies of data. In general, the time needed for a Bubble Sort varies roughly according to the formula $N^2/2$, meaning that increases in N are multiplied together. With the Shell-Metzner sort the variation in time roughly follows the pattern:

```
1.6*N*(LOG N/LOG 2).
```

What this means is that if, assuming the worst case, a Bubble Sort of 10 times were to take one second, 100 items would take 100 seconds and

1000 items would take 10,000 seconds.

When we turn to the Shell-Metzner Sort, again assuming the worst case, if a sort of 10 items took one second, then a sort of 100 items would require 20 seconds and one of a thousand items would require 300 seconds.

These are not meant to be accurate figures on the speed of the sorts on the 64 but they do illustrate the immense differences that arise when the amount of data to be sorted grows. You can prove it for yourself by playing around with the size of the list to be sorted in the test program.

```
1040 GOSUB 4000:TI$='000000':GOSUB 10000
:PRINT TI$:GOSUB 6000:GOSUB 7000
10000 REM*********************************
10001 REM SHELL-METZNER SORT
10000 REM*********************************
10010 GAP=2^(INT(LOG(IT-1)/LOG(2))+1)
10020 GAP=GAP/2
10030 C=IT-GAP-1 : B=0 : IF GAP<1 THEN R
ETURN
10040 A=B
10050 D=A+GAP : IF B$(A)>B$(D) THEN 1008
0
10060 B=B+1 : IF B>C THEN 10020
10070 GOTO 10040
10080 T$=B$(A)
10090 B$(A)=B$(D)
10100 B$(D)=T$
10110 A=A-GAP : IF A<0 THEN 10060
10120 GOTO 10050
```

In practice, running the Shell-Metzner on a list of 100 items, you will find a typical saving of only around 60 per cent compared to the Bubble Sort. Increasing the size of the list, however, will produce some more dramatic results. For a 200 item list, the Shell-Metzner may require only 15 per cent of the time needed for a Bubble Sort.

Conclusion

We have not begun to exhaust the topic of sorting. There are other sorts which can provide a significant, though not astronomical, saving over the Shell-Metzner. The problem with them is that they almost invariably require that extra storage space be set aside in order that data may be shuffled between the main list and one or more subsidiaries. For practical purposes this means that, for the kind of quantities where they really begin to show their paces compared to the Shell-Metzner, they are too

wasteful of memory to be of practical use on a home micro. Using the sorts given in this chapter you will be able to handle almost any quantity of data, from the smallest to the largest, with methods appropriate to the complexity of the task and, very often, with a dramatic increase in the speed of your programs.

CHAPTER 9
Data Structures: I

One of the axioms of this book is that most useful programs store data. Storing and processing data is what micro-computers are best at, an area where they far exceed the capabilities of any other method. Sometimes, however, the very ease with which a micro handles information can be an excuse for sloppiness in deciding just how that information is to be stored. Anything seems to go as long as it allows the program to keep the necessary data in memory.

The disadvantage of this kind of approach is that the wrong data structure can make an enormous hole in the abilities of a program, slowing it down, reducing the amount of material which can be used or unnecessarily complicating the structure of the working program. In this chapter we shall examine a few of the many ways of structuring data to fit the needs of the program and maximise speed and storage space.

Simple data structures: the custom-built array

By far the simplest of the structures which can be used to store any kind of information is an array whose dimensions are exactly fitted to the way in which the information itself breaks down. An example might be a program designed to record the turnover, profit, tax and investment figures for a series of companies. Here a numeric array might be declared, ARRAY (X,3), where X is the number of companies and data input with something like the following routine:

```
100 INPUT 'COMPANY NUMBER:';CY
110 INPUT 'TURNOVER:';ARRAY(CY,0)
120 INPUT 'PROFIT';ARRAY(CY,1)
130 INPUT 'TAX:';ARRAY(CY,2)
140 INPUT 'INVESTMENT';ARRAY(CY,3)
```

In a filing program to store names, addresses and phone numbers, a string array might be declared as ARRAY$(500,2), with inputs made by a routine similar to that given above.

Such structures can be as complex as the program requires. For

instance, it might be that for the company figures program mentioned above, the need was to record all of the figures over a five year period. In this case the array would be declared as ARRAY(X,3,4) and each of the separate categories would have a loop to allow five inputs, so that line 110 would become a routine in its own right:

```
110 FOR I=0 TO 4
112 PRINT 'TURNOVER FOR';1977+I;':' : IN
PUT ARRAY(CY,0,I)
114 NEXT I
```

The advantage of such tailor-made arrays is that they make the storing and retrieval of data immensely straightforward, everything has an unambiguous place and to find any item of data all you need to know is the number of the company, the category of the information, eg tax, and the year. Data in this kind of array is also easy to get at in new ways. For instance, if the user wished to know how each company had performed in terms of profit in 1978, a simple loop:

```
100 FOR I=0 TO ITEMS : PRINT ARRAY(I,2,1
) : NEXT
```

would extract the information without problems.

Another advantage of tailor-made arrays is the way in which whole sets of arrays can be matched together holding different but parallel sets of information. In the case of our example array it would be a simple matter to declare another array holding the names of the companies involved and these names could be accessed by exactly the same variable as was used to specify which set of figures are to be printed. In complex data-handling programs it is not uncommon to have a range of different arrays all storing parallel information. In cases where varieties of information are stored, much of it following similar structures, eg all relating to different months of the year, then tailor-made arrays can often be the only practical way of controlling the number of variables which have to be used to get information from many different places. If all the data were related to months of the year then in some cases a single variable representing the month in question would be enough to draw important information from every array.

Custom-made arrays, however, do have their disadvantages, among which is the amount of memory which even a fairly innocuous array can take up. A numeric array with dimensions of (100,10,10) might not seem very frightening in terms of size, but it would require 50,000 bytes of memory to hold it, far more than is available in BASIC on the 64. To calculate the amount of space the desired array will require, simply

multiply together all the figures used in the brackets when the array is declared and then multiple again by:

2 in the case of an integer numeric array
3 in the case of a string array
5 in the case of a floating-point numeric array

remembering that for string arrays, the figure arrived at is all 'overhead' and that anything put into the array will be added to the amount of memory required. In the case of the numeric arrays, all the elements are set up with zero in them, so there will be no increase in the size of memory required if elements are changed.

When declaring large and complex arrays, it is therefore vital to determine whether all the space in the array is going to be fully used. There are many occasions in the filing of information when a large number of categories need to be declared but not every item in the file is going to have information under each category. Such arrays are known as 'sparse' and the number of applications which can carry the kind of wasted memory that a complex but sparse array can create are fairly limited.

The conclusion to all this is that if you can set up a data structure which is exactly tailored to your information, and if it will hold all the information you are ever likely to want to store on the topic then go ahead and do it, it will without doubt simplify your eventual program immensely. Eventually, however, you will run into applications where the sheer number of items or the complexity of the structure is such that a tailor-made array is simply too wasteful to contemplate and you will need to go on and examine some of the other types of storage in this chapter.

DATA STRUCTURES FOR NUMBERS

Single byte numbers in integer arrays

The 64 provides a very economical method of storage for the range of numbers mostly used in data-handling programs. Most of the values used in such programs — not the values which may be stored but the values used for the purposes of controlling the program — are integer numbers (no decimal points) in a limited range reflecting the sizes of things like arrays and the length of strings. For most of these types of values, to use a normal numeric array is a sheer waste of memory, since every element of a floating-point array requires five bytes of memory compared to the two needed for one element in an integer array.

Integer arrays can, however, be used to reduce the amount of space necessary to store small numeric values even more by packing more than one value into each element of an array. This is possible because each

element in an integer array can store a number in the range –32768 to +32767, effectively a range of 65536. The range available simply represents what can be stored in a 16 bit number which uses the leftmost of the 16 bits to record whether the number is positive or negative — this is discussed more fully in the section on the FRE function in the chapter on strings. What is important to us is that one element in an integer array can hold a number up to 256*256–1 (when allowance has been made for the negative part of the range), so that if we take two numbers in the range 0–255, multiply one of them by 256 and then add the second, two numbers are effectively stored within one larger number. For instance, if we wanted to save 237 and 76, in that order, it could be done with a line of BASIC reading:

```
100 NN% = 256*237 + 76: REM ANSWER IS 60
748
```

Because the range of numbers which can be stored is not 0–65535 but –32768 to +32767, the number which will be stored in the array has to be altered if it is over 32767 by the subtraction of 65536, so that the necessary line of BASIC now becomes:

```
100 NN = 256*237 + 76 : IF NN>32767 THEN
 NN=NN-65536
110 NN% = NN
```

To decode a pair of numbers which has been stored in an element of an integer array in this way, another simple line of BASIC will suffice:

```
100 NN=A%(X) - 65536*(A%(X)<0) : N1 = IN
T(NN/256) : N2 = NN-256*N1
```

The advantage of storing numbers betweeen 0 and 255 in this way is that the memory savings are almost the same as storing numbers in the characters of a string (ie one byte per character with a very small overhead for the array itself) and in loading and saving to tape or disk the numbers can be stored straight, without having to translate between characters and numbers. The disadvantage of the method is that for normal usage it is actually twice as slow as translating from character values. In normal use you are unlikely to notice this but you will need to make a choice between the two methods of storing according to how much saving and loading you will be doing.

To save a number of values in the range 0 – 255, a routine such as the following could be used:

```
100 DIM A%(99)
110 INPUT 'POSITION: ';PP
120 INPUT 'VALUE: ';NN
130 TT=A%(PP/2)-65536*(A%(PP/2)<0)
140 IF PP AND 1 THEN TT=256*INT(TT/256)+
NN
150 IF NOT PP AND 1 THEN TT=(TT AND 255)
+256*NN
160 A%(PP/2)=TT+65536*(TT>32767)
170 N1=INT(A%(PP/2)/256) : N2=A%(PP/2)-2
56*N1
180 PRINT N1,N2
190 GOTO 110
```

Line 130 extracts the current value of the correct element of A% and deals with the problem of negative numbers. The correct position in A% line is determined by PP/2, so that the first two values will be stored in element zero, the second two in element one and so on. Note that there is no need to round down the numbers when dividing PP by 2 results in a value like 2.5, since the 64 will understand A%(2.5) as A%(2). In line 140 the 'IF PP AND 1' detects odd numbered items, since PP AND 1 will only be true for odd values of PP. If the value of PP is odd then the value to be inserted into the particular element of the array is the lower of the two values which will be stored in that element. We therefore preserve the upper value in the form of 256*INT(NN/256), thus dropping any part of the number which is less than a whole multiple of 256, then add the number to be stored, making it the lower of the two values which are to be kept in that element. In line 150 the 'IF NOT PP AND 1' detects even numbers, which will be placed into the higher position in their respective elements in the array. This is accomplished by ANDing TT with 255, thus preserving all the bits in NN which represent numbers below 128 and eliminating all the others. To the result is now added 256 times the value to be stored, this value becoming the higher of the two stored in the particular element. Line 160 readjusts TT to fit into the –32768 to +32767 range and replaces it into A%.

To unpack a structure like this is simpler than creating it in the first place:

```
200 FOR I=0 TO 99
210 NN=A%(I/2) - 65536*(A%(I/2)<0)
220 IF NOT I AND 1 THEN NN = INT(NN/256)
225 IF I AND 1 THEN NN = NN-256*INT(NN/2
56)
230 PRINT NN
```

```
240 IF I/10=INT(I/10) THEN GET T$:IF T$=
'' THEN 240:REM PRINT TEN AT A TIME
250 NEXT I
```

The advantage of saving memory in this kind of way is obvious though it should be stressed that it has its cost in terms of the time taken to store and extract values if they are constantly being accessed by the program. One advantage in terms of speed will be in loading and saving to tape or disk, where only half the number of elements need be processed.

Storing in free memory

For those who like or need to use every ounce of free memory for their programs, it should not be forgotten that the 64 holds in reserve 4096 bytes of memory which cannot be accessed in normal BASIC for the purposes of creating and storing variables. Beginning at memory address 49152 this area can, however, be used by means of BASIC POKE and PEEK statements, and almost any orderly structure of numeric or even string data can be stored there if you are prepared to take some care in working out the structure and the variables which will be used to control access to the data.

The easiest form of data to store in such a free area of memory is single-byte numeric values, ie numbers in the range 0–255. To simulate an array with dimensions of 50 by 50, for instance, items can be stored by means of a statement such as:

```
100 POKE 49152+50*X+Y,NN
```

where X is the number of the row in the array, counting from zero, Y is the number of the column, again counting from zero, and NN is the number to be stored. Values are retrieved by a mirror image of the statement, this time using PEEK:

```
100 NN=PEEK(49152+50*X+Y)
```

More complex array structures, with more than two dimensions, can be created provided that you observe the following rules:

1) Work out what your array would be if it were declared in BASIC, eg A(20,10,5).

2) Working from the right, calculate the number of elements in each unit of each of the dimensions. In the example given above there would be one element in each of the units of the rightmost dimension, five in each of those on the second from the right and 50 in the leftmost.

3) When specifying a location in the hypothetical array, first put the start location of the area of memory you are using, then multiply each dimension specified by the number of elements you have calculated and add to the start location. Thus element 14,7,3 in the example given would be location 49152 + 50*14+5*7+3.

More complex structures can be defined for numbers requiring two or more bytes, though you will always have to use techniques like those described in the previous section to divide the numbers into single bytes. In using two byte numbers, for instance, the rule would be to multiply the address obtained from the dimensions by two and then to POKE into the resulting address and the one following it the two bytes making up the number to be stored. To store a number in the range 0–65535 in location 14,7,3 of a two-byte per element array with dimensions as in the example above, the following routine would suffice:

```
100 N1=INT(NN/256) : N2=NN-256*N1
110 PP=49152+2*(50*14+5*7+3)
110 POKE PP,N1 : POKE PP+1,N2
```

Numbers in strings

We have already seen in Chapter 3 that storing data in variable length strings can be both fast and economical. Using this kind of method, new items of data can be added to the beginning, end or middle of a block of data, with all the other items stored automatically relocated. We have also seen that such data need not be limited to the maximum 255 bytes imposed by the length of a single string, since several elements of a string array can be used.

This discovery is not simply relevant to the question of storing small items of string data. Strings can often be very usefully employed in storing numeric values. The reason this is possible is that every character in the 64's character set has a unique value, called its 'ASCII value', a number in the range 0–255. It is in this numerical form that the 64 actually stores the characters. BASIC provides the programmer with two functions ASC and CHR$ which allow translations to be made between numbers and characters so that values can be translated into characters, stored in the form of a string and subsequently reconverted to numbers again.

The technique for transforming a value into a number is illustrated by the following line:

```
100 NN=65
110 A$=CHR$(NN)
120 PRINT A$
```

Run these lines and what will appear will be the letter A whose character value happens to be 65. Retranslation can be illustrated by adding the following line:

```
130 PRINT ASC(A$)
```

We can see from this that the function of CHR$ is to create a character with the code value specified, while ASC extracts the code value of an existing string character. We can combine the two to perform economical and flexible forms of storage for small numeric values:

```
100 A$=''
110 INPUT NN
120 IF NN >= 0 THEN A$=A$+CHR$(NN) : GOT
O 110
130 FOR I=1 TO LEN(A$)
140 PRINT ASC(MID$(A$,I))
150 NEXT I
```

This little routine will allow you to input up to 255 numbers between zero and 255. If you input a minus number the second half of the routine will come into play and print out the data in the order in which they were input. Note that in line 140, the MID$ function does not actually specify that the part of the string from which an ASC value is to be extracted is only one character long. MID$(A$,I) actually means the whole of A$ beginning at character position I. This can be done because the ASC function only ever works on the first character of any string it is presented with.

The usefulness of this technique is that string handling does not limit us to merely adding characters on to the end of an existing string. Line 120 could just as easily have been:

```
120 IF NN >= 0 THEN A$=CHR$(NN)+A$ : GOT
O 110
```

so that the routine would have placed the numbers in reverse order, something that would have been harder to do in a normal array without constantly having to shift the existing data up the array.

Equally, we could use some of the techniques drawn from Chapter 3 to insert values anywhere in the existing string:

```
100 A$=''
110 INPUT 'NUMBER TO BE INSERTED';NN
```

```
120 PRINT 'POSITION (1 TO';LEN(A$)+1;')'
; : INPUT PP
130 IF NN>=0 THEN A$=LEFT$(A$,PP-1)+CHR$
(NN)+MID$(A$,PP) : GOTO 110
140 FOR I=1 TO LEN(A$) : PRINT ASC(MID$(
A$,I)) : NEXT
```

When dealing with values which are greater than 255, there are no memory savings in using strings to store values but the flexibility of being able to insert values into the middle of existing data without shifting everything else can still be extremely useful. The following routine will insert a two byte number (0–65535) anywhere into a single string:

```
100 A$=''
110 INPUT 'NUMBER TO BE INSERTED';NN
115 N1=INT(NN/256) : N2=NN-256*N1
120 PRINT 'POSITION (1 TO';LEN(A$)/2+1;'
)'; : INPUT PP : PP=PP*2
130 IF NN>=0 THEN A$=LEFT$(A$,PP-2)+CHR$
(N1)+CHR$(N2)+MID$(A$,PP-1) : GOTO 110
140 FOR I=1 TO LEN(A$) STEP 2
150 PRINT 256*ASC(MID$(A$,I))+ASC(MID$(A
$,I+1))
160 NEXT I
```

Deletions can easily be accomplished using the techniques described in Chapter 3 and multi-string techniques from the same chapter can be employed to increase the capacity from that of a single string. This will be discussed in the section on pointer arrays later in this chapter.

From all of this it can be seen that storing numbers in strings is a real option for programs where values need to be inserted or deleted regularly. The main disadvantage here is that many of the characters, though they can be stored in memory as string characters, are non-printing characters, that is to say outside the range of characters which the 64 can normally print. This makes very little difference in relation to printing on the screen, since the characters themselves would be meaningless anyway — it is the ASCII values we are interested in. Where it does make a difference is in loading and saving, which must also employ a form of PRINTing, ie PRINT#. Unfortunately, the 64 is not capable of saving and loading non-printing characters as such — they must be converted to numbers and saved in that form. Thus for a string of characters, A$, storing numeric values, the saving routine would have to include:

```
100 PRINT#1,LEN(A$)
110 FOR I=1 TO LEN(A$)
120 PRINT#1, ASC(MID$(A$,I))
130 NEXT I
```

and the string would need to be recreated on loading by something like:

```
100 INPUT#1,LL
110 FOR I=1 TO LL
120 INPUT#1,TT : A$=A$+CHR$(LL)
130 NEXT I
```

This adds considerably to the time in loading and saving so a decision must be made whether to use string storage on the basis of how much saving and loading is to be done in comparison to the advantages of the method while the program is running.

Stacks

One application of the technique of storing numbers in strings is the creation of stacks. The principle of a stack is the same as that of the common-or-garden office spike. As letters come into the office they are placed on the spike then taken off and dealt with at a convenient time. Because of the way in which they are stored, the letters are always processed in the order 'last in — first out'.

Stacks are very important in computing since many of the operations a computer will perform are dictated by information stored in a data storage area known as a stack. Take the example of GOSUBs. In any chain of GOSUBs, the address to which program execution will RETURN is always that of the *last* GOSUB. Thus the address of each GOSUB is stored on the top of the stack and, when a RETURN is encountered, the address on top of the stack is the one acted upon.

In BASIC programs a stack can be used whenever several items need to be removed from a data structure at the same time and modified. As each is removed it can be placed in a separate array and its place in the main data array stored on a stack of two byte numerical values. When replacing the items in the main array their correct places can be obtained from the beginning of the stack.

More commonly, however, a stack is used to remember the positions of a group of items sharing some characteristic, especially groups of items which will be added to or subtracted from frequently. Later in this chapter we shall see how this can be applied to remembering how many empty spaces there are in an array and where they are. Each time an item is deleted from array, its address in the array is stored on a stack of two-byte values:

```
100 P1=INT(PP/256) : P2=PP-256*P1
110 SK$=CHR$(P1)+CHR$(P2)+SK$
```

where PP is the address of the item being deleted. To retrieve the address of an empty space in the array all that is necessary is:

```
100 PP=256*ASC(SK$)+ASC(MID$(SK$,2)) : S
K$=MID$(SK$,2))
```

This allows items to be inserted into an array without having to shift all the other items to get the empty spaces at the end. When SK$ is reduced to an empty string it is an indication that there are no further spaces available.

STRING DATA STRUCTURES

Packed strings

In the chapter on string handling we have already examined some of the simpler ways in which strings can be used to store information and how more can be packed into a smaller space. One example of this was the simple packed string using a separator character to identify the individual parts, so that the whole of an entry in a name and address file might be stored in the form:

```
SMITH*JOHN ANTHONY*11 THE STREET*ANYTOWN
*ANYCOUNTY*P00 0CO*0909 11111*MALE
```

The advantage of such a structure is that each item, instead of having an overhead of three bytes of memory if it had a complete element in a string array assigned to it, requires only one byte, the one containing the *. This may not seem like an enormous saving at first but for a file containing entries each having eight separate items, as in the example given above, that represents a saving of:

8*3 (for a separate element for each item) – 3 + 7 (three bytes for the single string plus seven separators)

or 14 bytes for a single entry. For a 500 entry file, the saving represented would be 7000 bytes, and that is a saving worth having when total memory available for program and data is less than 39000 bytes.

A drawback to such a method of packing strings is the time taken to access the individual items. In order to unpack the example given above, we need to use a search routine, as described in the chapter on string

handling, to identify the separate parts. Searching a 500 entry file for a particular item, especially if it fell towards the end of a string, could take a painfully long time. One way around this is to provide the information about the way in which the string is packed in a format which is easier to get at than searching through the strings for asterisks. Such a technique is known as using 'pointers' and we shall be examining later how to make much more complex and effective use of pointers in structuring a file but here the technique is a simple one of attaching, to the front of the string, numbers which indicate where the characters are meant to be broken up:

```
100 PTR$='' : IN$=''
110 FOR I=1 TO 8
120 INPUT TT$
130 IN$=IN$+TT$
140 PTR$=PTR$+CHR$(LEN(IN$))
150 NEXT I
160 IN$=PTR$+IN$
```

All that happens here is that, as each one of eight strings is input, the program takes the length of IN$, which is the overall entry and then adds a character with the same ASCII value as that length to the pointer string PTR$. At the end of the loop, all the items are stored in IN$, without any indication of where one ends and the next one begins, while in PTR$ are stored eight characters whose ASCII values record the endpoint of each of the items. These two are added together to produce the entry in the form in which it will be stored in memory.

Having stored the items in this way, a simple routine is all that is required to unpack them again:

```
200 P1=9
210 FOR I=1 TO 8
220 P2=ASC(MID$(IN$,I))+8
230 PRINT MID$(IN$,P1,P2-P1+1)
240 P1=P2+1
250 NEXT I
```

Here the values of the pointer characters at the beginning of the string are used to scan through the string, determining the start and end of each item. All that we need to know at the beginning of the routine is the number of pointer characters, so that the position of the first character of the first item can be determined (ie the first character after each pointer). From then on each pointer character tells us where an item ends and we know that the next item starts one character after. The particular routine given here assumes a regular structure of eight items, but this

need not necessarily be the case. Items could be input in any number up to the maximum string length and then an extra character added to the front of PTR$ which recorded how many items there were. This first character would then be used by the loop which picked up the items to determine how many pointer characters were to be expected.

This technique is considerably faster than expecting the program to scan through the string character by character looking for separators between items. Its main limitation is that it does limit the length of a single entry to 255 characters minus the number of characters required for the pointers. For most filing applications, however, 255 characters is more than adequate and where it is not the limitation can be overcome by using two or more packed strings for each entry.

Packed strings using numeric array pointers

Using packed strings in the way outlined above has the disadvantage that obtaining the ASCII value of individual characters can take time if large numbers of items are being processed. More serious than this is the fact that most of the characters created in the pointer section of the array will be non-printing characters. This makes no difference when they are stored in the memory, but the 64 is not capable of storing non-printing characters easily on cassette or tape. In order, for instance, to store a single packed entry in the format given above onto a cassette, a routine such as the following would be needed:

```
100 FOR I=1 TO 8 : PRINT#1,ASC(MID$(IN$,
I) : NEXT I
110 PRINT#1, MID$(IN$,9)
```

while to reclaim the data would require something like:

```
100 IN$="" : FOR I=1 TO 8 : INPUT#1,TT :
 IN$=IN$+CHR$(TT) : NEXT I
110 INPUT#1,TT$ : IN$=IN$+TT$
```

All this translation between numbers and characters during loading and saving takes time and this can be reduced by storing the pointers in a numeric array, in the manner indicated in the section on packing numbers into integer arrays. This can only be done when the number of items in each packed string is known and regular but can simplify saving and loading considerably. If you do want to store pointers in an integer array then what has to be done is to determine how many items you want to pack into each string and then dimension an integer array with each line half that number (plus one if the number is odd). Thus, to cope with a file of packed arrays containing 9 items each, an array would be declared

such as ARRAY%(500,4) and an input routine such as the following used:

```
100 DIM ARRAY%(500,4)
105 INPUT X
110 FOR I=0 TO 9
120 INPUT TT$
130 IN$=IN$+TT$
140 NN=ARRAY%(X,I/2)-65536*(ARRAY%(X,I/2
)<0)
150 IF I AND I THEN NN=256*INT(NN/256)+L
EN(IN$)
160 IF NOT I AND 1 THEN NN=NN-256*INT(NN
/256)+256*LEN(IN$)
170 ARRAY%(X,I/2)=NN+65536*(NN>32767)
180 NEXT I
```

The lines of this routine look fairly forbidding but they contain nothing we haven't already dealt with, in fact the routine mirrors the input routine in the section above on packing numbers into an integer array.

To unpack a structure like this is again more complex than simply using characters within the string as pointers:

```
200 P1=1
210 FOR I=0 TO 9
215 NN=AR%(X,I/2)-65536*(AR%(X,I/2)<0)
220 IF NOT I AND 1 THEN P2=INT(NN/256)
225 IF I AND 1 THEN P2=NN-256*INT(NN/256
)
230 PRINT MID$(IN$,P1,P2-P1+1)
240 P1=P2+1
250 NEXT I
```

You can run these routines together, inserting and recalling the items from a string since the value of 'X' is undefined and therefore will be zero. In normal use, some other part of the program would be dictating the place into which the new item was being placed.

Whether you want actually to use this method will depend upon the ratio between the amount of loading and saving to be done and the amount of processing you want to carry out on the individual items. Actually processing an item stored by this method, extracting the pointers or replacing them, takes approximately twice as long as the method using values stored in strings, though you are unlikely to notice this in processing any one item. It is nevertheless a useful addition to your armoury of methods and worth bearing in mind, if only for the sake of

variety. In the next chapter we shall go on to examine data structures which are considerably more complex to program but which can save considerably on time as well as memory.

CHAPTER 10
Data Structures: II

In the last chapter we looked at some neat and easy methods of storing data which allow a number of problems to be solved. In this chapter we turn to some data structures which, while they are no less useful in solving problems, take considerably more programming.

Linked lists

In what has gone before when discussing data structures for numbers, you will have noted that very often what we have been doing is employing techniques to reduce the amount of shifting data that has to be done. To insert a one-byte value in the first position of a numeric array involves shifting all the current data one space up the array. Using a string to store the values, as we have noted, results in everything else being shifted automatically when a new item is added to the beginning. Sometimes, however, for both numbers and strings, the data can only practically be stored in an array such as A$(500), where each entry will require one line of the array. On the assumption that the data starts at line zero in the array, then inserting a new item towards the beginning is going to involve shifting large quantities of data to make room for it, and this can take considerable amounts of time, as well as creating possible problems with garbage collection which we noted in Chapter 3.

The solution to this kind of problem is often to store the data in the order in which it was input and to keep a separate record of where each item would be if the array were arranged in the chosen order, such as alphabetical order for strings. In the case of a 'linked list', each item in the list will have attached to it the address in the array of the next item in the desired order. Take the example of a set of three string items: AAA, CCC and DDD. If we want to add a new item, BBB, in current alphabetical order, there is no need to move CCC and DDD to make room. All that we need to do is to somehow attach to the item AAA an indicator which tells the program that the next item in alphabetical order is not item two but item four. Having found item four there must be an indicator attached to *that* which tells the program that the next item is to be found at position two. Provided that the correct indicators are attached to each item, the program will process the items in the order one, four, two, three.

To demonstrate the method we shall use, cut yourself six small squares of paper about two inches in size and draw a line across the middle of each. The upper half of each square will be used to keep the pointer to where the list is going, the lower half to record the actual item to be stored. On one of the slips write '1' on the upper half (fairly small since it's going to be replaced), and 'START' on the bottom. On another slip write '65535' at the top and 'STOP' at the bottom. Put those slips next to each other with room for the remaining four slips next to them in a line. From now on we shall be inserting slips in alphabetical order and when comparing any new slip we shall count START as coming before it in the alphabet and STOP as coming after.

Now take another slip and write 'BBB' on the bottom half. Now compare BBB with START. According to the rule above, BBB comes after START so look at the top half of START and move on to the slip indicated (they are numbered from zero). Obviously the slip indicated is STOP and BBB should come before that so we have found the correct position for BBB — immediately after START and before STOP.

Now comes the important bit. Don't move the two existing slips, just put the BBB slip down in the third position (position two when counting from zero). Cross out the 1 from the top of START and replace it with '2'. START now points to the new slip BBB. On the top of the new slip write '1' since the next slip in alphabetical order is STOP. If you now follow the pointers on the top of the slips, beginning with START, the order is START, BBB, STOP.

Now take a fourth slip and write 'DDD' on the bottom. Follow the pointers through until you reach a point where one slip is less than DDD and one is more. In this case you will be after BBB and before STOP. Place DDD in the fourth position. Cross out the '1' on BBB and replace it with '3'. Put '1' on top of DDD to indicate that the slip after it is STOP. The pointers now give the order START, BBB, DDD, STOP.

You should now be able to deal yourself with the two remaining slips. one of which should be made 'AAA' and the other 'CCC'. At the end of the process the pointers should give the order START, AAA, BBB, CCC, DDD, STOP. Note that this has nothing to do with the order of the slips on the table, it is purely a reflection of the alphabetical order and it is achieved without ever having to move something that is already in place. If you have understood this you are now in a position to go on to look at the same method being carried out in BASIC. This time we shall work with strings in an array, using a parallel integer array to store the pointers.

The following routine will create a linked list in alphabetical order:

```
1000 REM*********************************
1001 REM LINKED LIST
1002 REM *********************************
```

```
1100 DIM A$(499), A%(499) : IT=2 : HO=0
1110 A$(0)=CHR$(0)
1120 A$(1)=CHR$(255)
1130 A%(0)=1
1140 A%(1)=32767
1150 INPUT 'ITEM TO BE INSERTED';IN$ : I
F IN$='STOP' THEN STOP
1160 ADD=0
1170 FOR I=1 TO IT-1
1180 TT=ADD
1190 ADD=A%(ADD)
1200 IF A$(ADD)<IN$ THEN NEXT I
1210 A%(TT)=IT
1220 A$(IT)=IN$ : A%(IT)=ADD
1230 IT=IT+1
1240 GOTO 1150
```

This routine takes some explaining, so we shall look at it line by line:

1100 A$ is the main array the data will stored in, A% will be used to store the pointers and IT represents the number of items already stored. This is initially set to two since the array will be set up with two dummy entries to mark the beginning and end of the linked list. The variable HO will be explained later.

1110–1140 These are the two dummy entries like START and STOP in our example. In the first position (address 0) in the arrays goes an entry whose pointer points to the second position (address one), with the content of the item being CHR$(0). This means that for all normal strings, this entry will always be the first item in the array in alphabetical order. The second position in the array has an entry whose link bytes point to 32767 (this value will never be used since the last item has nothing to point to) and whose main content is a single byte of value 255. This item will always fall at the end of the file in alphabetical order. The reason for these two entries is that like many data structures, it is easier to start off with what amounts to a working file than to put special conditions in the routine to test whether a new item falls at the beginning or end of the list. The first and last items of the file have to be treated differently since one has no item pointing forward to it and the other has no item to point forward to. Many methods of storing data have problems with first and last items and in many cases it is simpler to bypass the problem and put an artificial first and last line in when array is set up.

1160 The variable ADD will be used to store the position in the array of the item currently being compared: it starts at zero.

1170 The maximum number of times we shall have to make a comparison between the item input and an existing item is equal to the number of items in the array.

1180 TT will normally be the position of the item last examined in the alphabetical search. We need to remember it in case the new item needs to be inserted in between that item and the next, in which case the pointer of the item at TT will have to be changed.

1190 ADD is now set to the position indicated by the pointer for the item at TT, ie the next item in the linked list.

1200 If the existing item at ADD is not greater than the new item then we have not found the correct place, so the loop moves us on to the next item as indicated by the link bytes.

1210 If we have reached this point the correct position for the new item has been reached. The pointer for the item at TT must now be made to point to the position where the new item is to be inserted, ie position IT.

1220 The new item is inserted, with its pointer indicating the item which was previously pointed to by the item at TT.

1230 One is added to IT to reflect the fact that there is now one more item.

To check that the routine is working properly, enter the following lines, which will print out the list in alphabetical order:

```
1152 IF IN$='PRINT' THEN GOSUB 2000 : GO
TO 1150
2000 REM*********************************
2001 REM PRINT LINKED LIST
2002 REM*********************************
2010 ADD=0
2020 FOR I=1 TO IT-HO-2
2030 PRINT A$(A%(ADD))
2040 ADD=A%(ADD)
2050 NEXT I
2060 RETURN
```

Here the link byte of each item printed is used to obtain the address of the next. Once again the variable HO is used — it will be explained under the next routine.

Deletion follows the same kind of outline. This routine deletes a

numbered item from the list — it would be just as easy to specify the string and search through the list for it before deleting:

```
1154 IF IN$='DELETE' THEN GOSUB 3000 : G
OTO 1150
3000 REM*********************************
3001 REM DELETE ITEM
3002 REM*********************************
3010 INPUT 'NUMBER OF ITEM TO DELETE';NN
3020 P2=0
3030 FOR I=0 TO NN-1
3040 P1=P2
3050 P2=A%(P2)
3060 NEXT I
3070 A%(P1)=A%(P2)
3080 HO=HO+1
3090 RETURN
```

Note here that nothing is actually deleted, all that happens is that the pointer for the item before the one to be deleted is set equal to the pointer of the item to be deleted. The item before the one to be deleted now points to the item after the one to be deleted and the program will no longer be aware of the deleted item, even though it will still be there. If you did want to actually delete a line could be added:

```
3065 A$(A%(P1))=''
```

In this routine the variable HO will be incremented every time an item is deleted. This may seem strange since it would seem like common sense to simply knock one off the value of IT. The problem with this is that new items are always inserted at the end of the list, and IT indicates the end of the list. Because we are not actually changing the addresses of any of the items when inserting or deleting, reducing IT would mean that a new item would actually overwrite an existing one. Later in this chapter we shall examine how to deal with the fact that this leaves us with wasteful 'holes' in the array.

Linked lists are a useful method to employ in programs where lists have to be regularly compiled but they suffer the disadvantage that they can only be accessed from the end, in this case only one end, the beginning. It is possible to create lists which link both forwards and backwards by having two pointers for each item, one pointing to the item before and one to the item after the current item but you can still only sensibly start at one of the ends. If you jump into a linked list there is no way to determine how far along the list you are, you only know the address of a neighbouring item. To put it another way, if you want the 30th item in

a linked list there is no way just to jump to it, you must follow through the tortuous trail of 29 previous pointers. In the next section we shall examine a way around this problem.

Pointer strings

In the previous section we saw that it is possible to create an ordered list without having to constantly shift items around in an array. Items can simply be added to the end of the array and the task of determining their proper positions left to a separate array of pointers. The problem was that there was no way to jump into the list to determine the position of, say, the 30th item. We have, however, noted in the previous chapter and Chapter 3 that there is one type of data structure into which it is possible to insert items into their correct positions without shifting the existing items, and that is a string. We also know that it is possible to store numbers in strings and have experimented with techniques which allow us to do just what cannot be done with a linked list, that is to jump to any particular position in the string. Putting this together with what we have learned about pointers, we end up with the pointer string, a technique which will allow us to jump into the middle of an array yet still preserve the advantages of inserting new items without shifting the existing contents of the list to make room.

To do this we need only one new technique, and that is how to insert numbers in the range 0-65535 into a multi-string array where the total length of the strings involved varies. This is similar to a technique examined in Chapter 3 and can be achieved with something like the following routine:

```
4000 REM*********************************
4001 REM INITIALISE
4002 REM*********************************
4010 DIM PTR$(19) : IT=0 : HO=0
4020 FOR I=0 TO 1 : FOR J=1 TO 125 : PTR
$(I)=PTR$(I)+'01' : NEXT J,I : IT=250
5000 REM*********************************
5001 REM INPUT ITEM
5002 REM*********************************
5010 INPUT 'NUMBER (1-32767):';NN
5020 NN$=CHR$(NN/256)+CHR$(NN-256*INT(NN
/256))
5030 GOSUB 8000
5040 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2) + N
N$ + MID$(PTR$(LL),2*LP-1)
5050 IT=IT+1
5060 GOSUB 9000
```

```
5070 GOTO 5010
8000 REM*********************************
8001 REM OBTAIN POSITION
8002 REM*********************************
8010 PRINT 'POSITION (1 TO';IT+1+(NN<=0)
;'): ';:INPUT PP
8020 LL=INT((PP-1)/125) : LP=PP-125*LL
8030 RETURN
9000 REM*********************************
9001 REM RE-ARRANGE ARRAY FOR ADDITION
9002 REM*********************************
9010 FOR I=0 TO 18
9020 IF LEN(PTR$(I))<=250 THEN 9050
9030 PTR$(I+1)=RIGHT$(PTR$(I),2)+PTR$(I+
1)
9040 PTR$(I)=LEFT$(PTR$(I),LEN(PTR$(I))-
2)
9050 NEXT I
9060 RETURN
```

Note that line 4020 is a temporary line which is only used to test the routine.

Run the routine and enter the following values for number and position when prompted:

```
16705
251

16962
252

17219
253

16962
1

16705
1

16962
126
```

```
16705
126
```

Stop the program with RUN/RESTORE and you can now verify that the routine is working by entering directly, without a line number:

```
?PTR$(0)
```

which should result in a string of 01s preceded by AABB which the numbers 16705 and 16962 translate into when expressed as two-byte character pairs. Entering:

```
?PTR$(1)
```

should give the same result.
The real test is now to enter

```
?PTR$(2)
```

should give a string 01010101AABBCC. Counting the number of characters in the first two strings should reveal that there are six full lines on the screen plus another 10 characters. If all this checks out, what it shows is that the routine can accept two-byte numbers along the full length of the data and will automatically shift the remaining data so that no single string exceeds 250 bytes in length.

```
6000 REM*********************************
6001 REM DELETE FROM POINTER STRING
6002 REM*********************************
6010 PRINT 'DELETE ';: GOSUB 8000
6020 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2) + M
ID$(PTR$(LL),2*LP+1)
6030 GOSUB 9500
6040 RETURN
9500 REM*********************************
9501 REM RE-ARRANGE ARRAY FOR DELETION
9502 REM*********************************
9510 FOR I=0 TO 18
9520 IF LEN(PTR$(I))>=250 OR LEN(PTR$(I+
1))=0 THEN 9550
9530 PTR$(I)=PTR$(I) + LEFT$(PTR$(I+1),2
)
9540 PTR$(I+1)=MID$(PTR$(I+1),3)
9550 NEXT I
9560 RETURN
```

To delete items we need only add a slightly edited version of the insert routine, plus another line to allow you to jump to this second part:

```
5015 IF NN<=0 THEN GOSUB 6000 : GOTO 501
0
```

Now run the whole routine and enter:

```
16705
1

16705
126

16705
251
```

This will place AA at the beginning of the first three elements of the string array. Enter 0 in response to the next prompt for a number and when asked for the number to be deleted, answer '1'. Stop the program with RUN/RESTORE and you should find that PTR$(0) and PTR$(1) contain an array of 01s terminated by AA, and PTR$(2) contains 0101. This shows we can both insert and delete, adjusting the length of each element of the array as we go. You can now delete line 4020.

We now know enough to use a pointer string; all that remains is to give it something to point to. In the following routine, strings are added to an array in alphabetical order. The catch is that, instead of simply scanning the strings from position zero through to the end of the useful items, the strings will be examined, as each new item is entered, in the order dictated by the pointer string. The routine is as follows:

```
4000 REM*********************************
4001 REM INITIALISE
4002 REM*********************************
4010 DIM PTR$(19),A$(499) : IT=2 : HO=0
4020 A$(0)=CHR$(0)
4030 A$(1)=CHR$(255)
4040 PTR$(0)=CHR$(0)+CHR$(0)+CHR$(0)+CHR
$(1)
5000 REM*********************************
5001 REM INPUT ITEM
5002 REM*********************************
5010 INPUT 'ITEM TO BE INSERTED:';IN$
5014 IF IN$='STOP' THEN STOP
5016 IF IN$='PRINT' THEN GOSUB 7000 : GO
TO 5010
```

```
5020 GOSUB 8000
5030 NN$=CHR$(IT/256)+CHR$(IT-256*INT(IT
/256))
5040 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2) + N
N$ + MID$(PTR$(LL),2*LP-1)
5050 A$(IT)=IN$
5060 GOSUB 9000
5070 IT=IT+1
5080 GOTO 5010
7000 REM*********************************
7001 REM PRINT LIST
7002 REM*********************************
7005 IF IT-HO=2 THEN RETURN
7010 FOR PP=2 TO IT-HO-1
7020 LL=INT((PP-1)/125) : LP=PP-125*LL
7030 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
7040 PRINT A$(PA)
7050 NEXT PP
7060 RETURN
8000 REM*********************************
8001 REM OBTAIN POSITION
8002 REM*********************************
8010 FOR PP=1 TO IT-HO
8020 LL=INT((PP-1)/125) : LP=PP-125*LL
8030 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8040 IF A$(PA)<IN$ THEN NEXT PP
8050 RETURN
9000 REM*********************************
9001 REM RE-ARRANGE ARRAY FOR ADDITION
9002 REM*********************************
9010 FOR I=0 TO 18
9020 IF LEN(PTR$(I))<=250 THEN 9050
9030 PTR$(I+1)=RIGHT$(PTR$(I),2)+PTR$(I+
1)
9040 PTR$(I)=LEFT$(PTR$(I),LEN(PTR$(I))-
2)
9050 NEXT I
9060 RETURN
```

What the routine at 8000 does is to scan the array in the order dictated by the pointer array, with the correct address of each item in array A$ being obtained in the variable PA. When the correct position has been

discovered it is already stored in the variables PP, LL and LP and these can be returned to the routine at 5000, which is already set up to accept the desired position defined by those three. The only important change to the routine at 5000 is that NN$, the two characters which hold the new pointer until it is inserted, is a translation of IT, which is both the number of items so far stored and the number of the first free element in the array A$.

The working of the two routines together can be tested by entering PRINT when asked for an input. This will call up the routine at 7000 to print out the whole of the list, using successive values contained in PTR$.

Deleting with pointers

Having solved the problems of insertion, we can now tackle that of deletion when using a pointer array. We already have a routine to delete individual items from a pointer array. We shall adapt this so that instead of requiring the user to specify the number of the item, the user can input the item itself and have it searched for through the array, then the item and the pointer will be deleted. Add the following lines to the routine above:

```
5012 IF IN$="DELETE" THEN GOSUB 6000 : G
OTO 5010
6000 REM*********************************
6001 REM DELETE ITEM
6002 REM*********************************
6010 INPUT "ITEM TO BE DELETED:";IN$ : G
OSUB 8000
6020 IF IN$<>A$(PA) THEN RETURN
6030 PTR$(LL)=LEFT$(PTR$(LL),2*LP-2)+MID
$(PTR$(LL),2*LP+1)
6040 GOSUB 9500
6050 HO=HO+1
6060 A$(PA)=""
6070 RETURN
9500 REM*********************************
9501 REM RE-ARRANGE ARRAY FOR DELETION
9502 REM*********************************
9510 FOR I=0 TO 18
9520 IF LEN(PTR$(I))>=250 OR LEN(PTR$(I+
1))=0 THEN 9550
9530 PTR$(I)=PTR$(I) + LEFT$(PTR$(I+1),2
)
9540 PTR$(I+1)=MID$(PTR$(I+1),3)
9550 NEXT I
9560 RETURN
```

The same method used when inserting an item is used to find the correct place, namely the list is scanned in the order dictated by the pointer string for the first item that is greater, alphabetically, than the new item to be input. On return from this search routine, the item found in the list is compared with the item to be deleted and deletion is only carried out if the two are the same. If the item specified is found then its position is already in the variable PA, with the position of its pointer stored in LL and LP.

You should now be able to delete items which have been entered.

The black hole problem

One final problem remains to be solved, and that is what to do with the spaces created in an array when items are deleted. If we do not do this the array will eventually become riddled with such spaces until they take over completely and there is no more room for data, even though the array may be almost empty. This can be overcome by a method that we have already considered in outline, and that is the use of a stack. Still working with our pointer array routine we can insert a new line:

```
6065 ES$=CHR$(PA/256)+CHR$(PA-256*INT(PA
/256))+ES$
```

Thus every space created will be recorded in ES$ and all we need to do is to persuade the insert routine to take notice of the contents. This can be done by changing a few lines in the previous routine:

```
5022 TT=IT : IF LEN(ES$)=0 THEN 5030
5024 TT=256*ASC(ES$)+ASC(MID$(ES$,2))
5026 ES$=MID$(ES$,3)
5028 IT=IT-1 : HO=HO-1
5030 NN$=CHR$(TT/256)+CHR$(TT-256*INT(TT
/256))
5050 A$(TT)=IN$
```

Having entered these lines, whenever a new item is entered it will always be placed into the first available empty space in the array. Only if there are no spaces will the item be added to the end.

Conclusion

If you have worked your way through this chapter and the one before, you will no doubt realize that the point of it is not that you are going to

sit down tomorrow and use all the methods in your next program. The data structures described here are a reminder that there are many ways to use the memory of your 64, all of them with different strengths and weaknesses but all of them with a contribution to make in some circumstance or other. If you begin to program seriously there will come a time, probably several times, when a particular set of data, maybe only 10 or 20 items used at one point in a program, will baffle you. It is then that you will turn to this chapter and, hopefully, realise that one of the many methods presented here is the answer you needed all along.

CHAPTER 11
Inserting Data

Rather than explain too many issues at one time we have so far largely ignored the question of how, when new data is inserted into an array, it is to be done as quickly as possible and how the correct place is to be found efficiently. In this chapter we shall examine how to shift large arrays of data, how to conduct a fast search and how to tie in such a fast search to the techniques of pointer arrays examined in the last chapter.

Normal search and shift

By far the simplest way, in programming terms, to insert an item into a large ordered array is to search the array item by item and to move all the items which follow the correct place for insertion one place down the array. It is often said that the most efficient way to achieve this is to search from the end of the array, comparing with the item to be inserted and then shifting each item if the comparison reveals that in the end it is going to have to be moved to find space for the new item earlier in the array. Whether this is true or not depends upon whether it is possible to combine the lines which are necessary to search for the correct place for an item with those which move the data ready for the new item to be inserted.

The following listing is of a simple routine which first sets up an ordered array of numbers, then searches through the array for the correct place to insert a new item, then shifts data in order to make room for the new item:

```
500 REM**********************************
501 REM CONTROL ROUTINE
502 REM**********************************
510 GOSUB 750
520 TI$='000000'
530 GOSUB 1000
540 GOSUB 1500
550 PRINT TI$
560 STOP
750 REM**********************************
```

```
751 REM INITIALISE
752 REM**************************
760 DIM A%(9999) : IT=9950
770 FOR I=0 TO IT-1 : A%(I)=I : NEXT
780 INPUT 'NEW VALUE:';NI
790 RETURN
1000 REM*****************************
1001 REM SEARCH
1002 REM*****************************
1030 IF IT=0 THEN 1070
1040 FOR I=0 TO IT-1
1050 IF A%(I)>=NI THEN SP=I : I=IT-1
1060 NEXT I
1070 RETURN
1500 REM*****************************
1501 REM INSERT
1502 REM*****************************
1505 IF IT=0 THEN 1540
1510 FOR I=IT TO SP+1 STEP-1
1520 A%(I)=A%(I-1)
1530 NEXT I
1540 A%(SP)=NI
1550 RETURN
```

Note that this is a generalised routine, one that can be applied to other sets of data. For that reason we use IT to remember how many items there are as new items are entered. It is worth pointing out that in all such routines it is necessary to take account of the special case which arises when there are no items in the array. Unfortunately the loop 'FOR I=0 TO IT–1' will still execute once, even when it actually means 'FOR I=0 TO –1' and thus should not logically be executed. Allowing the loop to operate with these values will produce an illegal quantity error so the loop has to be jumped around.

To test the routine, enter 5000 when prompted for a new item. Printing out TI$ should show that this routine takes 81 seconds to execute (TI$='000121'). Using a search from the other end of the array, however, allows us to dispense with one of the loops, as in the following routine, which should be added to the one above:

```
560 TI$='000000'
570 GOSUB 2000
580 PRINT TI$
590 STOP
2000 REM*****************************
```

```
2001 REM SEARCH AND INSERT COMBINED
2002 REM*****************************
2010 IF IT=0 THEN SP=0 : GOTO 2060
2020 FOR I=IT-1 TO 0 STEP-1
2030 IF NI>=A%(I) THEN SP=I : I=0 : GOTO
 2050
2040 A%(I+1)=A%(I)
2050 NEXT I
2060 A%(SP+1)=NI
2070 IT=IT+1
2080 RETURN
```

This routine takes only 73 seconds to run when tested on the insertion of 5000 (TI$='000113') — a saving of some 10% over the previous method, as you will discover for yourself if you run the two routines together. You would get very different results if the item to be inserted were near the beginning or the end of the array, but putting an item into the middle gives us the average time that will be taken to input a series of items which are not weighted towards the beginning or end of the array.

The drawback to the second method is that it runs together two different program functions, the search and the insert. This is all very well for the present case, but it means that if we find a faster way of inserting or searching we shall not easily be able to patch it in to the program. We have already seen that there are faster ways of inserting items in the chapter on the use of pointer arrays. But there is also a much faster way of searching.

Binary searching

In the two routines given above it is clear that roughly 5000 comparisons have to be made before the correct position is found. Of course the array will not always have all the elements full of useful items. The number of comparisons, however, will always on average be equal to half the number of items in the array. The fact is that this is entirely unnecessary, since to find the correct place in the array above requires only 13 comparisons, maximum. Consider the following example:

1) In the array above we wish to insert a new item, the number 6172.

2) We begin the search for the correct position by examining the position in the array which represents the highest power of two which will fit into the total number of items in the array. In this case we have 10,000 items, and the highest power of two which will fit into that number is 8192 and

we shall call that the 'step value' or SV. Our first comparison is made between the new item and the item which is already in the array at the SV position. We shall call the search position SP, and to begin with it has to be made into 8191 to take account of the fact that the array starts at zero, not one.

3) The value at position 8191 (SP) is greater than the new item 6172, so we take the original SV of 8192 and divide by two, giving a new SV of 4096. This is subtracted from 8191 (SP), giving a new SP of 4095.

4) A comparison is now made at 4095 (SP). The number at that position in the array is less than the new item so we divide SV by two (=2048) and this time add it to SP, giving a new SP of 6143.

5) Once again, the item at 6143 is less than the new item so SV is divided by two (=1024). This is added to SP, giving 7167.

6) The item at 7167 is greater than 6172 so SV is divided by two (=512) and subtracted from SP, giving 6655.

7) The search is continued in the following locations and with the following jumps:

6655 (–256)
6399 (–128)
6271 (–64)
6207 (–32)
6175 (–16)
6159 (+8)
6167 (+4)
6171 (+2)
6173 (–1)
6172 the correct position.

Note that in all of this all we have ever done is add or subtract decreasing powers of two depending on whether the target position was up or down the array. Success did not depend on having an array, as in this case, of numbers increasing in steps of one. All that is needed is an array, string or numerical, which has been regularly ordered, either from low to high or high to low.

The number of comparisons which will have to be made to insert a new item into an array will, in general, be:

```
INT(LOG(IT)/LOG(2))+1
```

What this formula does is to say how many digits there are in the number ITems when it is expressed in binary notation, plus one, and this will always be the maximum number of comparisons to be made. Thus:

```
INT(LOG(500)/LOG(2))+1  =  9
```

and 500 in binary is:

111110100 – nine digits long.

Such a method clearly represents a massive saving when compared with the type of search used above, 4087 comparisons are saved. If we were searching for an item in an array, rather than a position in which to put something, then the saving might be even greater. Searching for an item might involve, if the item were not present, looking through the whole 10,000 items — with the binary search we would still use only 13 comparisons and if on the 13th comparison we had not found the desired item we would know that it was not in the array. In the following routine the binary search is used to insert an item at 5000, as in the previous listings:

```
590 TI$='000000'
600 GOSUB 3000
610 GOSUB 4000
620 PRINT TI$
630 STOP
3000 REM*********************************
3001 REM BINARY SEARCH
3002 REM*********************************
3010 IF IT=0 THEN SP=0 : GOTO 3090
3020 POWER=INT(LOG(IT)/LOG(2))
3030 SP=2^POWER-1
3040 FOR SV=POWER-1 TO 0 STEP-1
3050 SP=SP+2^SV*((NI<A%(SP))-(NI>A%(SP))
)
3060 IF SP>IT-1 THEN SP=IT-1
3070 NEXT SV
3080 IF A%(SP)<NI THEN SP=SP+1
3090 RETURN
4000 REM*********************************
4001 REM INSERT
4002 REM*********************************
4010 IF IT=0 THEN 4050
```

```
4020 FOR I=IT TO SP+1 STEP-1
4030 A%(I)=A%(I-1)
4040 NEXT I
4050 A%(SP)=NI
4060 IT=IT+1
4070 RETURN
```

There are two slight differences from the description of the method given above. Firstly, it is possible for the search pointer (SP) to jump above the number of items in the array, which finish at IT–1. If this happens, line 3060 resets SP to the top of the data and this change of value makes no difference to the operation of the search. Secondly, if the item input is not in the array already, the position found will be one of the two values which would be above or below it if it were there. If the value found would have been above it then all the data from that point can be shifted to make room. If the value below is found then SP must be moved one up the array before being used to say where the shifting of data will end and this is done by line 3080.

Running the routine again on the value 5000 will produce a timing of only 49 seconds, which is basically the time taken to shift the elements. The delay caused by the high number of comparisons has been almost entirely eliminated.

Pure searching

In the examples given above, it was assumed that we were searching in order to insert a new item. This need not of course be true. Methods one and three can be used simply to find the position of an item, if indeed it exists within an array. To see how this can be done, alter the control routine to read:

```
500 REM*********************************
501 REM CONTROL ROUTINE
502 REM*********************************
510 GOSUB 750
520 TI$='000000'
530 GOSUB 1000
540 PRINT SP,A%(SP)
550 PRINT TI$
590 TI$='000000'
600 GOSUB 3000
610 PRINT SP,A%(SP)
620 PRINT TI$
630 STOP
```

Inputting a value now will result in that value and its position being printed, together with the time by a straight search and the time by a binary search. Input 5000, the value in the middle of the array, and it should take 34 seconds to find by the straight search and one second by the binary search. Try putting in a non-integer number — this will return either the integer above or below the input value. By inserting a comparison such as:

```
615 IF A%(SP)<>NI THEN PRINT 'NOT PRESEN
T' : STOP
```

an automatic test for the presence of an item is provided.

Binary searching with pointer arrays

We have already seen that using the binary search eliminates all but a fraction of the time taken to search through an array for the position of a new item, leaving only the time taken to move the data when an insertion is made. We can now eliminate most of that delay by combining the binary search with the pointer array techniques learned in Chapter 10. The way in which we do this illustrates one of the strengths of writing the program in modules, for to change the method of searching for the correct position all we have to do is change one module, ensuring that it still returns the correct variable to the main part of the program. What happens in the search module is a matter of indifference to the rest of the program as long as the correct variable is returned. The module at 8000 in the pointer array program at the end of the last chapter now becomes:

```
8000 REM*********************************
8001 REM OBTAIN POSITION
8002 REM*********************************
8010 TT=IT-HO
8020 POWER=INT(LOG(TT)/LOG(2))
8030 PP=2^POWER
8040 FOR SV=POWER-1 TO 0 STEP -1
8050 LL=INT((PP-1)/125) : LP=PP-125*LL
8060 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8070 PP=PP+2^SV*((IN$<A$(PA))-(IN$>A$(PA
)))
8080 IF PP>TT-1 THEN PP=TT-1
8090 NEXT SV
8100 LL=INT((PP-1)/125) : LP=PP-125*LL
```

```
8110 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8120 IF A$(PA)<IN$ THEN PP=PP+1
8130 LL=INT((PP-1)/125) : LP=PP-125*LL
8140 PA=256*ASC(MID$(PTR$(LL),2*LP-1))+A
SC(MID$(PTR$(LL),2*LP))
8150 RETURN
```

This is clearly not a routine you will wish to enter for a small program which stores only small amounts of data. With large amounts, however, using a pointer array to cut out shifting of arrays and a binary search to reduce searching time can create an astonishing difference in speed.

CHAPTER 12
Odds and Ends

In this chapter you will find a number of unrelated techniques which do not, in themselves, justify a chapter but which are nevertheless worth bearing in mind when developing a program. The techniques covered are: DEF statements, DATA statements, FOR loops, timing with TI/TI$ and rounding with INT.

User defined functions

A function in BASIC is an instruction which performs a complete operation of some kind on a number or string to produce a desired result. Most of the functions on the 64 are mathematical, they perform a mathematical operation on a number to produce something like the sine of that number. Without these built-in functions, each time the user required a sine, cosine, tangent or the like, the full instructions for the calculations to produce them would have to be spelled out in BASIC.

Functions are extremely useful in saving program space and complexity for the particular calculations that they cover. Unfortunately they can't cover everything and there may be many occasions in a program where the same calculation has to be performed. Sometimes this can be coped with by use of a subroutine, and even a one-line subroutine. A single subroutine, however, can only work on one set of variables, thus:

```
1800 X=Y^2 + Y^3 + 5*Y
```

will only ever work on the variables Y and X. If you wanted to work on the variable Z and obtain the result in NN, then you would have to call the subroutine with a line like:

```
200 Y=Z : GOSUB 1800 : NN=X
```

A more flexible facility can be obtained by defining a new function with a line such as:

```
100 DEF FNA(Y)=Y^2 + Y^3 +5*Y
```

This line defines a function called FNA (it could have been FN followed by any valid variable name) with the option to replace the variable Y with any other in the formula during the course of the program. For instance the line:

```
300 NN=FNA(Z)
```

would accomplish everything that line 200 above did.

This facility can be used to simplify considerably the appearance of programs. In Chapter 11, for instance, when performing a binary search in combination with a pointer string, the line:

```
PA=256*ASC(MID$(PTR$(LL),2*LP-1))+ASC(MI
D$(PTR$(LL),2*LP))
```

was used three times. This wasteful repetition, with its scope for errors in entry, could have been avoided by the definition of a function when the program was initialised, eg:

```
100 DEFFNA(LL)=256*ASC(MID$(PTR$(LL),2*L
P-1))+ASC(MID$(PTR$(LL), 2*LP))
```

In the binary search module, all that would have been necessary would have been to include lines such as:

```
8060 PA=FNA(LL)
```

Here, no variable is replaced, so the function operates in exactly the same way as the original line. Many of the routines given in this book could be shortened by the use of defined functions.

Terminating FOR loops

You may have noticed during the course of this book that FOR loops are often used to scan data and return the position of an item in an array or string. One question that arises when using a loop in this way is the method by which the loop is terminated. This demands some thought because, as mentioned in Chapter 2, every loop declared requires some of the 64's limited 'stack' space, and this is only cleared when the loop is terminated properly with the loop variable reaching the upper limit set in the FOR statement. All this can be done by using a loop such as:

```
100 FOR I=0 TO IT-1
110 IF A$(I)=IN$ THEN PP=I : I=IT-1
120 NEXT I
```

Here, the moment the correct item is found in the array A$, the loop is terminated by changing the value of I so that it reaches the upper limit over which the loop is intended to operate. But now consider the following loop:

```
100 FOR I=0 TO IT-1
110 IF A$(I)=IN$ THEN GOTO 140
120 NEXT I
130 PRINT 'ITEM NOT FOUND' : FOR I=1 TO
5000 : NEXT
```

Here the loop will print 'ITEM NOT FOUND' if IN$ is not present in the array but will jump around the end of the loop and the error message if IN$ *is* present. This leaves the loop started in line 100 unfinished and consumes some space on the stack. The reason that I am prepared to use the technique despite this drawback can be illustrated by running the following short routine:

```
100 FOR I=1 TO 1000
110 FOR J=1 TO 10
120 NEXT I
```

Here the J loop is opened no less than 1000 times without being closed, with no dire results. The reason for this is that opening a loop with the same loop variable does not occupy extra stack space once the initial loop has been declared.

You can capitalise on this fact by using the same loop variable names continually. I almost always use loop variable names beginning with I and proceeding through the alphabet if more than one loop has to be open at the same time. Keeping to the same names, especially for loops which may not be properly terminated, means that I can enjoy the advantages of unfinished loops without getting mysterious 'OUT OF MEMORY' errors when the stack is full of the details of unfinished loops.

DATA statements

When a program works on a relatively fixed set of data there is often no point in going to the trouble of declaring a special array and then loading the information into it, the whole thing can be accomplished by DATA statements. DATA statements can be used for a variety of purposes in the same program. The only problem is that many people seem to get inordinately confused when they have to use different sections of data at different times, not knowing how to get to the right part of the data when they need it.

What lies behind this difficulty is the historical basis of the DATA statement, which goes back beyond the birth of the microcomputer to the earliest mainframes. In Chapter 4 we have already noted the difficulties which were often faced by programmers of mainframe computers before the development of 'interactive' computing where the user can supervise the execution of the program as it runs. Programs on such machines, like any other programs, needed data to work on, and this was provided in the form of punched cards. The cards were placed, in a stack, into a card reading machine and, when the program came to a statement such as 'READ NN', the card reading machine translated the holes punched in the first card into a number, took that number to be the value of the variable NN and then picked up the second card. There were only two ways through the stack of data cards, to examine the next card (whether you did anything with it or not), or to move to the beginning of the stack of cards. In the earliest of micros it was recognised that to be able to specify data in a program was a useful facility but that facility was written in as if it was still a matter of reading cards. The BASIC program was allowed to contain statements such as:

```
100 DATA 12,45827,67,123,76593,212,1065
```

and the items of data in such lines could be read by lines such as:

```
200 READ X
```

There were still, however, only two ways to move around within the data provided. For the first execution of line 200 above, the variable X would be set to 12, the first item of data, the second READ instruction would set a variable equal to 45827 and so on. The only way to break this sequence would be to use another command:

```
300 RESTORE
```

which would move the 'data pointer' back to the beginning of the first line of data in the program.

Many modern micros have gone beyond this limitation by introducing the command RESTORE LN. LN is a line number and the data pointer would be set pointing to the first item of data in the first data statement at or after line number LN in the program. In this way, several different sections of data could be included, each serving a different purpose in the program, and information could be picked up from each section at different times without difficulty. Unfortunately, the 64 does not include this extra facility, and ways have to be found around the limitations imposed.

An example of the most straightforward use of DATA statements might be something like the following:

```
1000 DIM MO$(11), AA%(9)
1010 DATA JANUARY,FEBRUARY,MARCH,APRIL,M
AY,JUNE
1020 DATA JULY,AUGUST,SEPTEMBER,OCTOBER,
NOVEMBER,DECEMBER
1030 DATA 12,125,23,64,17,176,38,78,169,
5
1040 DATA INITIALISE,INSERT,DELETE,PRINT
,STOP
1050 FOR I=0 TO 11 : READ MO$(I) : NEXT
1060 FOR I=0 TO 9 : READ AA%(I) : NEXT
1070 FOR I=1 TO 5: READ T$ : PRINT T$ :
NEXT
```

Here the data is being used to load one array with the names of the months and another with some necessary data. The third section of DATA is being used to print out the options on a menu rather than use individual PRINT statements for each. This all works very well the first time it is run, but what if we need to print the menu a second time later on during the use of the program. The only thing to do, with the data statements as they stand is reset the data pointer to the beginning of the data and read through the data again, ignoring the bits that are not needed — the month names don't need to be loaded again. This would require something like:

```
2000 FOR I=1 TO 22 : READ T$ : NEXT
2010 FOR I=1 TO 5: READ T$ : PRINT T$ :
NEXT
```

In other words data can be read and discarded, provided that the start position of the desired data is known exactly. Unwanted data is read in the form of strings, because this will accept either number or string data. When there are many DATA statements, however, it is easily possible to make mistakes in calculating the position of particular groups of data, so that the wrong information is being READ by the program. More importantly, any change in the number of data items will throw the whole process out of synch, necessitating going through the program and changing the values of all the loops which read and discard data items. This is tiresome at the best of times but it becomes completely impractical for those programs which are designed to run on DATA statements which are changed fairly frequently.

The way around this is to structure DATA statements with indicators which tell the program where it is within the DATA or when the data has ended. Applying this to the routine above we get:

```
1000 DIM MO$(11), AA%(9)
1010 DATA #1,JANUARY,FEBRUARY,MARCH,APRI
L,MAY,JUNE
1020 DATA JULY,AUGUST,SEPTEMBER,OCTOBER,
NOVEMBER,DECEMBER
1030 DATA #2,12,125,23,64,17,176,38,78,1
69,5
1040 DATA #3,INITIALISE,INSERT,DELETE,PR
INT,STOP,#-1
1045 FOR I=1 TO 1000 : READ T$ : IF T$='
#1' THEN I=1000
1046 IF T$='#-1' THEN RESTORE
1047 NEXT I
1050 FOR I=0 TO 11 : READ MO$(I) : NEXT
1055 FOR I=1 TO 1000 : READ T$ : IF T$='
#2' THEN I=1000
1056 IF T$='#-1' THEN RESTORE
1057 NEXT I
1060 FOR I=0 TO 9 : READ AA%(I) : NEXT
1065 FOR I=1 TO 1000 : READ T$ : IF T$='
#3' THEN I=1000
1066 IF T$='#-1' THEN RESTORE
1067 NEXT I
1070 FOR I=1 TO 5: READ T$ : PRINT T$ :
NEXT
```

What has been done here is to tell the program to search through the DATA statements for the marker which begins the correct section of data and only then to begin picking up information. The number 1000 in the new loops is simply any number greater than the number of items of data present in DATA statements. If the program READs to the end of the data, instead of generating an 'OUT OF DATA' error it detects the '#–1' marker and knows that it must return to the first DATA statement to continue READing. You would not, of course, want to use a technique such as this for the small amount of data given above but for programs which hold a lot of data in this way, structuring DATA statements can be a boon.

Timing with TI and TI$

In the routines which have gone before you will no doubt have noted that two very different methods are employed to dictate and to obtain timings during the course of a program. When dictating a span of time, perhaps for the display of a particular message on the screen, FOR...TO loops are almost invariably used, eg:

```
100 PRINT 'THIS IS A MESSAGE'
110 FOR I=1 TO 2000 : NEXT
120 PRINT '[CLR]'
```

This is a straightforward technique which anyone can use, arriving at the necessary value for the loop by trial and error. The method, however, breaks down when more complex actions are to be timed. It is all very well to print something (a single action) and then to pause for a time using a loop, but what if you wanted to go on performing a *series* of actions for a time and then terminate those actions when the time was up. This can sometimes be achieved by use of a loop but it is far easier to employ the 64's built-in timing functions, TI and TI$.

These two are what is known as 'system variables', each having their respective values as variables but these values normally being set not by the user but by the 64 itself. TI is a numeric variable which is set to zero when the 64 is first switched on and then incremented by one every sixtieth of a second. The number of seconds which has elapsed since 'power up' can therefore always be obtained by simply entering:

```
? TI/60
```

Probably more useful in everyday terms is TI$, which is derived from TI but is in a more comprehensible format *and* is capable of being set by the user to any desired value. TI$ starts off at '000000' when the 64 is switched on and is incremented every second. The value displayed, however, is not simply the number of seconds but HOURS/MINUTES/SECONDS, so that 021537 means, two hours, 15 minutes and 37 seconds. The comparison between the two variables can be easily seen by entering:

```
? TI/60,TI$
```

TI$ can be set to a desired value in the same way as any other string, with the value of TI following suit even though TI cannot be altered directly. Thus entering:

```
TI$='120000'
```

would set it equal to 12 hours. The main limitations here are that an error

is generated if the number to which TI$ is set equal does not have the correct six digits and if a value greater than '240000' is used, TI$ is actually reset to zero.

We have already used this feature of TI$ on several occasions to obtain the comparative of different methods of sorting and searching, first setting TI$ equal to '000000' and then, when the process being timed was finished, printing it out again to see how much time had elapsed. There are a variety of ways in which even this technique can be used, giving reaction times on tests or games or perhaps adding a professional touch to an applications program by telling the user how long the program has been in use when a particular session has been terminated.

In addition, TI and TI$ can easily be used to *dictate* timings rather than merely to display them. Take the following two lines:

```
1000 TT=TI
2000 IF TI-TT>18000 THEN RETURN
```

If the first of the two were placed at the beginning of a particular routine and the second line embedded somewhere in the routine where it would be accessed regularly, then the routine in question would be terminated after five minutes (18000/60 = 300 seconds = 5 minutes). If there is no need to preserve the value of TI then even the task of calculating the desired number of sixtieths of a second can be avoided. These two lines would serve the same function:

```
1000 TI$='000000'
2000 IF TI$>'000500' THEN RETURN
```

The one fly in the ointment here is the use of the cassette deck, since TI and TI$ are not incremented while it is being used.

A 'real-time clock' feature can be added to a program by slicing TI$ up and re-presenting it slightly more clearly:

```
1000 TT$=TI$
1010 PRINT LEFT$(TT$,2);':';MID$(TT$,3,2
);':';MID$(TT$,5)
```

The use of an intermediary string, TT$, is preferred to working directly on TI$ since it is possible that TI$ may change while the slicing is going on. Thus if TI$ were '005959' when the hours were being printed and then changed to '010000' before the minutes and seconds were extracted then the resulting time would be '00:00:00'.

Rounding with INT

In the routines that have gone before the INT function, which strips a number of any decimal places, has often been used. We have seen that INT can be extended to dictate the maximum number of decimal places a number can have by a manipulation such as:

```
100 NN=INT(1000*NN)/1000
```

which would give three places.

It should not be forgotten that INT can also be used to round numbers to the *nearest* integer rather than always the one below. This is accomplished by simply adding 0.5 to the number and then subjecting it to INT. Test the following routine by entering a range of non-integer numbers:

```
100 INPUT NN
110 PRINT INT(NN+0.5)
120 GOTO 100
```

CHAPTER 13
Formatting

The ability of the 64 to place data on the screen under the control of a program is a curious mixture of great flexibility and, at the same time, a lack of some fairly standard features available on most of the current generation of micros. The flexibility arises from the ability of the programmer to use the straightforward cursor control characters to move the print position around, to control colour characteristics with other control characters, and the ease with which characters can be poked into the screen memory and hence onto the screen. The lack is of commands like PRINT AT, which would allow a particular position on the screen to be specified with a single BASIC command, or PRINT USING, which permits the format of numbers or strings to be specified in such a way, for instance, as to ensure a standardised number of decimal places in printing. In this chapter we shall examine how to make the most of the 64's strengths and find ways of avoiding some of its weaknesses.,

Cursor controls

There are as many ways of using cursor control characters as there are different uses of the screen to present information. Programs which create complex structures on the screen are probably more easily accomplished on the 64, because of the cursor control characters, than on any comparable popular home micro. Whole screens of information can be laid out without the constant recalculation of position necessary with PRINT AT. Items can be moved up, down, left or right in relation to the last item with almost trivial ease. Take the following example, which will print a line of four-character strings diagonally across the screen from bottom right to top left:

```
100 A$='****'
110 PRINT '[CLR][24*CD][34*CR]';
120 FOR I=1 TO 24
130 PRINT A$;'[CU][5*CL]';
140 NEXT
```

Two conclusions can be drawn from this little example. Firstly that

moving one item in relation to the last, that is to say a 'relative move' is straightforward and logical. You simply define the number of spaces in any direction you wish to move and use the appropriate cursor control. The second conclusion is drawn from line 110 and it is that cursor controls can become cumbersome for defining start positions which are not close to the top right hand corner of the screen. This limitation can be overcome somewhat by defining two strings at the initialisation of the program eg:

```
100 HO$='[39*CR]'
110 VE$='[24*CD]'
```

These two strings contain enough cursor control characters in the (VE)rtical and (HO)rizontal dimensions to take the print position anywhere on the screen, starting at the top right. To get to a print position on the bottom line would now require only something along the lines of:

```
120 PRINT '[HOME]';LEFT$(VE$,24);LEFT$(H
O$,34);
```

This looks longer on the page of this book than the original, but of course to specify the strings in full would have required one and a half lines. If you want to use what is effectively a version of PRINT AT regularly, you can shorten the process somewhat by writing a subroutine to take over the task:

```
1710 PRINT '[HOME]';LEFT$(VE$,VE);LEFT$(
HO$,HO);:RETURN
```

Having entered this, the print position can be sent anywhere on the screen with a line such as:

```
120 VE=12 : HO=12 : GOSUB 1710 : PRINT '
HELLO'
```

This is slightly shorter than the previous version using LEFT$ each time and is clearer to read. One thing to remember is that the screen positions here are numbered not from one but from zero, so that vertical runs from 0–24 and horizontal from 0–39. If you want to work from one as the starting point for each then the subroutine must be modified to work for VE–1 and HO–1.

Using this method we can try the diagonal line of strings again (remember that this assumes HO$ and VE$ have been defined):

```
120 A$='****'
130 FOR VE=24 TO 0 STEP-1
```

```
140 HO=VE+10
150 GOSUB 1710 : PRINT A$;:REM The semi-
colon is to stop scrolling
160 NEXT VE
170 END
```

This points out the strength of PRINT AT, even if it is only an imitation, and that is that variables can be used to format a screen. Quite complex structures can be created in this way. The following routine will create a triangular table of two-character strings. In this case the string is a dummy but the information could easily be drawn from an array of important data:

```
120 A$='**'
130 FOR VE=0 TO 12
140 FOR HO=0 TO 3*VE STEP 3
150 GOSUB 1710 : PRINT A$
160 NEXT HO,VE
170 END
```

PRINT AT can also be used to allow the user to control the formatting of a screen and this can be a very useful facility. We might, for instance, have a 12 by 6 array containing information which is presented on the screen in the form of a table. PRINT AT can be used to allow the user to input new items and see them placed on the screen without having to reprint the whole screen from the array. With the formatting strings VE$ and HO$ defined and the subroutine entered, a typical routine might be:

```
500 DIM A%(11,5) : PRINT '[CLR]'
510 FOR VE=0 TO 11 : FOR HO=0 TO 30 STEP
 6 : GOSUB 1710
520 PRINT A%(VE,HO/6) : NEXT HO,VE
530 VE=21 : HO=0 : GOSUB 1710
540 INPUT 'ROW';VE
550 INPUT 'COLUMN';HO
560 INPUT 'VALUE (0-32767):';NN
570 A%(VE,HO)=NN : HO=HO*6
580 GOSUB 1710 : PRINT '      ' : REM FIV
E SPACES TO CLEAR POSITION
590 GOSUB 1710 : PRINT NN
600 GOTO 530
```

When you run this routine you will find that there is one problem, and that is that the INPUTs are always printed over their predecessors, which

can be confusing, and that brings us to another useful formatting string, which I usually call O$ in my own programs. O$ consists of nothing more than 39 spaces and is, once again, defined at the start of the program. Having done this it can be used in conjunction with PRINT AT to clear lines which are used repeatedly. In the routine given above, a loop could be added:

```
525 HO=0 : FOR VE=21 TO 23 : GOSUB 1710
: PRINT O$ : NEXT
```

and any characters on the lines to be used would be erased.

Use of cursor control characters

Despite the usefulness of a simulation of PRINT AT, it should not be assumed that using the cursor control characters directly in PRINT statements is ruled out. Most screens are not regular in their structure but are laid out so that they are clear for the user. PRINTs and INPUTs will most often be spaced out to attract the eye or to divide information into logical sections on the screen. Small relative moves of the print position are best dealt with by the use of the cursor controls:

```
100 INPUT 'VALUE 1:';A
110 INPUT '[CD]VALUE 2:';B
120 INPUT '[CD]VALUE 3:';C
130 INPUT '[3*CD]VALUE 4:';D
140 INPUT '[CD]VALUE 5:';E
```

To use the PRINT AT subroutine to space out inputs in this way would be wasteful in the extreme. Even some regular structures are better achieved by use of the control characters direct. An example of this might be the printing of a word vertically down the screen rather than across. Here, the inclusion of two cursor controls saves a great deal of messing about with variables:

```
100 A$='HELLO'
110 FOR I=1 TO LEN(A$)
120 PRINT MID$(A$,I,1);'[CD][CL]';
130 NEXT I
```

Equally, when overprinting text on the same line, there is no point in being over-clever when a simple up cursor will do:

```
100 PRINT 'THIS IS TEXT'
110 FOR I=1 TO 2000 : NEXT
120 PRINT '[CU]AND THIS WILL OVERPRINT I
T'
```

In the chapter on input we saw that the same technique could be applied to INPUTs, starting the prompt with an up cursor so that, if an error was made in what was entered, the INPUT could be printed again and would automatically relocate itself over the previous characters.

TAB

Another aid to formatting a screen is the TAB function, which allows data to be spaced into separate columns. This little routine prints groups of characters across the screen, evenly spaced:

```
100 PRINT '[CLR]'; : FOR I=0 TO 36 STEP
3
110 PRINT TAB(I);'**';
120 NEXT I
```

Clearly this could easibly be used to format a table on the screen, though one or two things need to be noted. Firstly the format of TAB is TAB(X) *without* a space between the TAB and the open bracket character. If you put a space between the two, and it looks natural to do so, the data will not be formatted and a '0' will appear before each item. Secondly, the TAB always works from the beginning of the line on which printing is currently taking place and only from left to right. Thus TAB(10) means 'print from character position 10 provided that character position 10 has not already been passed'. If you try to TAB to a position *before* the current position on the line, TAB will effectively be ignored and the item will be printed in the next available print space. You can see this for yourself by changing the routine above:

```
100 PRINT '[CLR]'; : FOR I=36 TO 0 STEP
-3
110 PRINT TAB(I);'**';
115 GET T$ : IF T$='' THEN 115
120 NEXT I
```

Running the routine now results in the '**' print position always moving to the right as if the TAB did not exist. Try removing the semi-colon from the end of line 110 and you will see that the problem does not arise when moving to a new line each time.

A feature of TAB is that it does not clear the lines it passes over, only the character positions that are being printed on. This means it can be used for changing tables in a similar way to the PRINT AT simulation above:

```
100 FOR I=1 TO 10 : PRINT '*** '; : NEXT
```

```
 :REM 1 space after asterisks
110 PRINT '[CU]';
120 FOR I=0 TO 32 STEP 8 : PRINT TAB(I);
'XXX'; : NEXT
```

SPC

Parallel to TAB is the SPC function, which is used to move the print position a number of characters to the right compared with the current position. The literature that comes with the 64 describes space as a function which prints spaces but this is incorrect. Like TAB, SPC prints nothing, it simply moves the position at which printing will be done without disturbing any characters in the intervening area. You can test this for yourself by entering this little routine:

```
100 PRINT '[CLR]XXXXX'
110 PRINT '[HOME]';SPC(7);'*'
```

The line of Xs which SPC passes over is unaffected.

Because SPC works relative to the current print position it is likely to be less useful in formatting a regular structure on the screen than TAB unless the items it is printing will all have a standard length. Separating items on a series of lines with SPC will result in them being equally spaced but they will not necessarily be in columns. One strength of SPC, however, is that many printers are unable to deal properly with TAB but work perfectly with SPC. If you have this problem in outputting tables then it is a matter of defining the width of the column you wish to print, including spaces, then using SPC to make up any difference between the length of the string to be printed and the column width:

```
100 A$(0)='*' : A$(1)='**' : A$(2)='***'
 : A$(3)='****'
110 OPEN 1,4 : CMD 1
120 FOR I=1 TO 10
130 FOR J=0 TO 3
140 PRINT A$(J);SPC(6-LEN(A$(J)));
150 NEXT J : PRINT : NEXT I
160 PRINT#1 : CLOSE 1
```

In this routine lines 100 and 160 open a file to the printer, send all printed output to that file for a while and then close the file again when printing is finished. Note also that a PRINT on its own has to be used at the end of each line of items. Unlike TAB, which will ignore the semi-colon at the end of the PRINT statement when the end of the line is reached, SPC would go on printing six character columns over the end of the line and

thus spoil the formatting.

Simulating PRINT USING

When formulating tables it is often advantageous to be able to work with items of a standard length of format. To illustrate why, run the following routine:

```
100 PRINT '[CLR]';
110 FOR I=1 TO  20
120 PRINT (RND(0)*15)^2
130 NEXT
```

and see what a mess the resulting column looks. There would be little point in producing a table with that kind of format, even though everything actually starts at the same position on successive lines. Useful presentation of numeric data relies on formatting the data so that different values can be compared at a glance.

Many micro computers overcome this problem by means of a command called 'PRINT USING' which allows the programmer to specify a format for a number or string — how many digits, how many decimal places, should it be padded out with spaces to a standard length. Unfortunately this is not available on the 64 but, like PRINT AT, most of the necessary features can be easily simulated.

To format a number we shall first translate it into a string. This will not affect the original value, it is a simply a temporary measure for the purpose of printing which allows us to get information on the length of the item and to manipulate the format. Using a simple property of logarithms we shall detect how many places there are before the decimal point and then employ simple string handling to standardise the format. This is best done with a subroutine, since it is likely that formatting may be needed at several points during the program:

```
100 SS$='                    ' : REM AS MAN
Y SPACES AS NECESSARY
1810 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))+STR$(NN)
1820 RETURN
```

Line 100 is simply a reminder that this routine will only work if a string of spaces has already been declared as SS$. The reason the routine works is that the expression LOG(NN)/LOG(10) returns the value of NN when expressed in logarithms of base 10. This is useful for the simple reason that the integer part of any logarithm with base 10 is one less than the number of digits before the decimal point. Thus LOG(120)/LOG(10) is 2.0918125, and adding 1 to the integer part (the '2') tells us that the

number has three digits before the decimal point. In fact, as you will no doubt have noted, for the purposes of this particular calculation we take the LOG of ABS(NN), since the LOG of a negative number is a nonsense. Armed with such information it is a simple matter to add to the front of the number enough of SS$ to ensure that there are always a standardised number of characters before the decimal point. This is done using LEFT$ and STRS$, the latter being the function which turns a number into a string with the same appearance as the original number. In the case of the routine above there will normally be nine spaces or digits before the decimal point. The reason that there are nine, rather than eight is that STR$ adds a space to the beginning of a positive number where a minus sign would be in a negative number.

You should now be able to demonstrate the function of the subroutine using a modified version of the random number generator given previously:

```
100 PRINT '[CLR]';
110 FOR I=1 TO 20
120 NN=(RND(0)*15)^2
130 GOSUB 1810
140 PRINT NN$
150 NEXT
160 END
```

Having solved the problem of aligning the decimal points, there is another problem to be dealt with before some data can be fitted into a strict table format, and that is the number of places *after* the decimal point. This needs to be tackled in a slightly different way. Numbers with too many decimal places can be dealt with by simple string slicing but it is not quite so simple for those which have too few. We cannot simply pad out our formatted string with zeros to the required length, since some of the numbers we wish to deal with may not have a decimal point, and adding zeros to them will make a nonsense of the result. Fortunately we can avoid having to make tests of the number of decimal places by the simple expedient of adding to each number a decimal fraction *beginning with at least as many zeros as we require in our standardised format.* Thus to get two decimal places we need to do no more than to add .001 to the original number. When the number is transformed into a string it will be a simple matter to slice off the final '1'. This will make no difference to the value of numbers which already have two decimal places but will add trailing zeros and, if necessary, the decimal point to those which do not. The only complication is the need to make some special provision for negative numbers. This kind of formatting can be achieved by another short routine such as:

```
1900 REM*************************************
1901 REM FORMAT DECIMAL POINTS
1902 REM*************************************
1910 NN$=STR$(INT(100*NN)/100+(.001*SGN(
NN)))
1920 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))    LEFT$(NN$,LEN(NN$)-1)
1930 RETURN
```

Only one obscure feature here and that is the use of the SGN function in line 1910. What SGN does when applied to a number is to produce a result of one or minus one according to whether the number is positive or negative. In line 1910 this clearly means that .001 will be added or subtracted from the result of the first half of the line according to whether the original number was positive or negative. The reason for this is that while 123 + .001 produces a useful result, 123.001, which can be used once the final '1' is removed, – 123 + .001 would give the totally useless result of –122.999, which is not at all what we want. The use of SGN ensures that if NN had been – 123 then the .001 would have been multiplied by –1 and therefore subtracted rather than added, giving the result –123.001, which *is* what we want.

We are now almost at the point where we have the means to control the format of a number within the limits required by an orderly table of data on the screen. One problem remains, however, and that is the possibility of a minus sign. It has already been pointed out that when a number is transformed into a string it has a leading space added to it if it is positive and a minus sign if it is negative. If we want to cram a great deal of data into a table we might well want to do without the possibility of a minus sign so that, for instance, if four spaces a number were allocated, we could deal with values up to 9999 rather than being stuck with a maximum of 999 plus an empty space where a minus sign might sometimes occur. The simple answer to this problem is to dispense with the minus sign, which does not stand out on the screen anyway, and use the 64's colour capability to mark out negative numbers:

```
2000 REM ********************************
2001 REM HIGHLIGHT NEGATIVE VALUES
2002 REM ********************************
2010 NN$ = STR$(INT(100*NN)/100+(.001*SG
N(NN)))
2020 NN$=LEFT$(SS$,8-INT(LOG(ABS(NN))/LO
G(10)+1))+LEFT$(NN$,LEN(NN$)-1)
2030 IF NN<0 THEN NN$=""+NN$+""
2040 RETURN
```

The extra line at 2030, compared to the previous routine, ensures that any negative value is not only formatted but printed inverse on the screen, thus ensuring that it stands out in the table. If you regularly use a colour monitor then you could perhaps colour the string red by placing the red control character at the beginning rather than the RVS character.

By using adaptations of these techniques you can now build up complex tables on the screen in orderly formats which will convey the information you desire far more effectively than ragged columns of figures whose meaning may not even be clear after two or three readings.

Justifying

When presenting strings as part of a table it can often be advantageous to give them a standardised length so that several columns can be printed with all the words starting or ending on the same character position in successive lines. This can be done by being clever with the print position but one easy method which can often be used is to pad out each string to a standard length. To pad out a string with spaces at the end to a total length of, say, ten characters, all that is needed is:

```
100 A$=LEFT$(A$+SS$,10)
```

where SS$ is the string of spaces used before. To pad out at the beginning (so that each word will end at the same place):

```
100 A$=RIGHT$(SS$+A$,10)
```

Both of these rely on the fact that string addition can take place within the brackets of a string function, thus saving us the trouble of having to construct a line such as:

```
100 A$=A$ + LEFT$(SS$,10-LEN(A$))
```

but note that both of the shorter versions will truncate A$ if it is longer than the allocated space. In some cases this may even be exactly what you want to keep the format of your table neat.

Easy logos and designs

Brightening up a good program with some simple graphics, a title page when the program is first run, some ornamentation here and there, can make the world of difference to the way the eventual user of the program perceives it. Fortunately for owners of the 64, it is perhaps the easiest of all the home micros on which to embellish a program, though it appears from the presentation of many programs that most owners do not realise

this. While on other home micros it may be necessary to go through complex planning to get a good design onto the screen, first planning it out on graph paper, then going through a laborious process of transferring the design into print statements. On the 64 all that is necessary is to treat the screen as if it were a sketchpad, using the outstanding character set to create the design of your choice *and then* to place that design easily into program lines.

For the simplest designs which do not need more than 35 character positions across, simply clear the screen and then move the cursor around, drawing in or erasing characters from the keyboard, text or graphics it doesn't matter. The only rule is to set yourself a margin at least five characters in from the left-hand side of the screen and at least one space on the right. Then, when you have created a design on the screen in this way, go back to the left-hand side of the screen and move down the screen putting in line numbers followed by a question mark and an 'open quotes' mark — you don't have to bother about closing the quotes.

When you have reached the bottom of your design, giving each line of characters a number, you will have effectively transformed your design into a program. It may be that you will want to play around with the positioning of your design on the screen but that is all you need to do. Anything which can be designed out of the normal character set can be input in this way, a picture, a title page of large letters, an attractive heading for a menu, even the outline of a complex table.

The main limitation here is that of the width of the design that can be entered in this way. If you want to put something on the screen which is wider than would allow you to put in the line numbers and '?' print abbreviation, you will have to design half a screen and then insert spaces at the beginning of the lines using SHIFT/INST. As you do this, each individual line will run over onto the next and the design will appear to have been destroyed. Don't worry about this, simply insert a line number at the beginning of each full line (not the spill-over parts) and, when you run the program you will find that the design is properly reprinted. The reason that only half the design is entered to begin with is that, as lines spill over the design will expand down the screen and any lines pushed off the bottom will be lost. Having entered half the design you can print all or part of it on the top half of the screen and add to it by the same method.

The possible need to insert line numbers is the reason why the final space of a line should be left free. If the design continues to the end of the screen in two consecutive lines the 64 will assume that it is intended to be a complete 80 character line and will not permit you to insert any more characters. If you need to go right to the end of the screen then this will have to be done once the design has been transformed into program lines. You can then go to the end of each of the lines and add the extra character necessary to each, followed by a closing quotation mark and

a semi-colon, otherwise the design will be printed with alternate blank lines.

Conclusion

There are few things more infuriating than a potentially useful program which cannot actually be used in any practical way for the simple reason that the data it presents is a mess. It takes very little effort to apply some of the techniques described in this chapter and no effort at all to see the difference that they make. Try it and see.

Postword

If you have worked through this book, entering all the techniques as you went, then you have more patience than I do. Of course, I *have* entered them, but then I'm paid to.

More likely you have skimmed through large chunks of the book, stopping to try out routines and techniques that seemed particularly appealing. Having done that, don't put the book away on the shelf and regard it as another text book to be referred to from time to time. The best way to use this book, if you don't want to plunge into some grand new project right away, is to call up some of your own programs and to see how what you have learned could be applied to them. It may seem strange, applying some of the more complex techniques to programs which are quite simple and even satisfactory, but that is not the point. The way to make any technique in computing your own is to go ahead and use it once or twice, even if the use isn't useful (if you see what I mean).

Another idea would be to take the routines in this book, even the one-liners and begin a dictionary of useful techniques on tape or disc. The more complex techniques, as you have already found, are already presented in such a way that you can enter and test them. One line techniques could have an input statement and a print statement added that would demonstrate their effect. Storing the routines in this way would allow you to examine them at a later date to see if they are the answer to a particular problem and to merge them into a program. as suggested in Chapter 1. As you read magazines and other books, more likely looking routines can be added, even though you don't want the programs of which they are a part. I have my own collection of routines which do nothing particularly useful at the moment (but do it rather well) and I am always grateful that I started it when a new challenge arises.

So don't forget the techniques in this book that you can't see a need for at the moment. Next week they may mean the difference between success and failure.

Other titles from Sunshine

SPECTRUM BOOKS

Master your ZX Microdrive
A Pennell — **£6.95**
ISBN 0 946408 19 X

The Working Spectrum
D Lawrence — **£5.95**
ISBN 0 946408 00 9

Spectrum Adventures
A guide to playing and writing adventures
T Bridge & R Carnell — **£5.95**
ISBN 0 946408 07 6

Spectrum Machine Code Applications
D Laine — **£6.95**
ISBN 0 946408 17 3

COMMODORE 64 BOOKS

The Working Commodore 64
D Lawrence — **£5.95**
ISBN 0 946408 02 5

Commodore 64 Machine Code Master
D Lawrence & M England — **£6.95**
ISBN 0 946408 05 X

Commodore 64 Adventures
M Grace — **£5.95**
ISBN 0 946408 11 4

Business Applications for the Commodore 64
J Hall — **£5.95**
ISBN 0 946408 12 2

Graphic Art for the Commodore 64
B Allan — **£5.95**
ISBN 0 946408 15 7

Mathematics on the Commodore 64
C Kosniowski — **£5.95**
ISBN 0 946408 14 9

ELECTRON BOOKS

Graphic Art for the Electron
B Allan **£5.95**
ISBN 0 946408 20 3

Programming for Education on the Electron Computer
J Scriven & P Hall **£5.95**
ISBN 0 946408 21 1

BBC COMPUTER BOOKS

Functional Forth for the BBC computer
B Allan **£5.95**
ISBN 0 946408 04 1

Programming for Education on the BBC computer
J Scriven & P Hall **£5.95**
ISBN 0 946408 10 6

Graphic Art on the BBC computer
B Allan **£5.95**
ISBN 0 946408 08 4

DIY Robotics and Sensors for the BBC computer
J Billingsley **£6.95**
ISBN 0 946408 13 0

DRAGON BOOKS

The Working Dragon
D Lawrence **£5.95**
ISBN 0 946408 01 7

Dragon 32 Games Master
K & S Brain **£5.95**
ISBN 0 946408 03 3

The Dragon Trainer
A handbook for beginners
B Lloyd **£5.95**
ISBN 0 946408 09 2

Advanced Sound & Graphics for the Dragon
K & S Brain **£5.95**
ISBN 0 946408 06 8

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, VIC 20 and 64, ZX 81 and other popular micros. Only 35p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £8.00 in the UK and £14.00 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role playing games. Include reviews of all the latest software, lists of all the software available and programming advice A year's subscription (12 issues) cost £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01 437 4343

This is a book for anyone who wants to begin real programming with the Commodore 64. It sets out to analyse some of the techniques required for the writing of successful applications programs.

The book is packed with advice and programming examples from single lines to more complex routines that will revolutionise your programming.

Using the techniques in this book you will find that you can write programs that are better, faster, clearer, more secure and more memory efficient than ever before.

Advanced Programming Techniques builds on the success of David Lawrence's previous best seller The Working Commodore 64 and is a response to the requests of many of you for a book that allows you to release the power of your micro in your own programs.

David Lawrence is one of the most successful microcomputer authors in Britain. He has a string of best sellers to his name. Besides writing books he is also a commercial software author and a regular contributor to Popular Computing Weekly.

SUNSHINE

ISBN 0 946408 23 8

£5.95 net



This was brought to you

from the archives of

http://retro-commodore.eu