ADVENTURE GAMES FOR THE COMMODORE 64



A. J. BRADBURY

Adventure Games for the Commodore 64

Other Granada books for Commodore 64 users

Business Systems on the Commodore 64
Susan Curran and Margaret Norman
0 246 12422 9

Commodore 64 Computing
Ian Sinclair
0 246 12030 4

Commodore 64 Disk Systems and Printers
Ian Sinclair
0 246 12409 1

The Commodore 64 Games Book
Owen Bishop
0 246 12258 7

Commodore 64 Graphics and Sound
Steven Money
0 246 12342 7

Software 64: Practical Programs for the Commodore 64
Owen Bishop
0 246 12266 8

Introducing Commodore 64 Machine Code
Ian Sinclair
0 246 12338 9

40 Educational Games for the Commodore 64
Vince Apps
0 246 12318 4

Adventure Games for the Commodore 64

A. J. Bradbury

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

Copyright © 1984 by A. J. Bradbury

British Library Cataloguing in Publication Data
Bradbury, A. J.
Adventure games for the Commodore 64.
1. Computer games 2. Commodore 64 (Computer)-Programming
I. Title
794.8'028'5404 GV1469.2

ISBN 0-246-12412-1

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

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 or otherwise, without the prior permission of the publishers.

Contents

Acknowledgements		vi
Important Note to the Reader		vii
1	Once Upon a Time ...	1
2	Plotting Your Adventure	10
3	Who Goes There?	20
4	O.K. Bugsy – We Know You're in There!	51
5	Interior Decor – Arrays and Things	71
6	One Step at a Time	106
7	A Code in Time Saves ...	125
8	The Well-chosen Word	145
9	Taking Shape	161
10	Sound and Vision	172
11	What Now?	188
Appendix: The Case of the Missing Adventure		197
Index		211

Acknowledgements

I would like to offer my sincere thanks to the following people who each, in their own way, contributed to the preparation and success of this book:

To Geof Wheelwright and Richard King of *Personal Computer News* for their encouragement when I first began to write about computer adventures.

To Perry, Nigel and Mike at Lion House, Brighton, for all kinds of invaluable assistance.

To my editors, Richard Miles and Allan Scott, who had the unenviable task of extracting the original manuscript from me and managed to turn it into a finished book.

And finally, and most sincerely, to my wife, Carol, for her unfailing support and patience.

A. J. Bradbury

Important Note to the Reader

All of the programs in this book were developed and tested on a Commodore 64 *before* being LISTed for publication. If you have any trouble getting a program to work on your own machine please check that you have typed in *exactly* what you see in the original listing.

The programs have all been written and LISTed using the C64's 'start-up' mode – *upper-case* only. To ensure that you are using the right mode please type in PRINT CHR$(142) – without a line number – before trying to enter any of these programs.

You might also like to start all your programs with a 'mode control' instruction to the computer to ensure that it RUNs the program correctly. This line should read PRINT CHR$(142) if you are using upper-case only, or PRINT CHR$(14) for mixed upper and lower case.

To make it easier for you to understand and use the listings, all programs have had bracketed instructions inserted in place of the usual inverse graphics symbols – as shown in the following table:

Listing	Press key(s)	Displays as:	
[CLEAR]	SHIFT CLR/HOME	♥	(inverse heart)
[HOME]	CLR/HOME	S	(inverse 'S')
[UP]	SHIFT UP/DOWN	○	(inverse circle)
[DOWN]	UP/DOWN	Q	(inverse 'Q')
[LEFT]	SHIFT LEFT/RIGHT	▮	(inverse bar)
[RIGHT]	LEFT/RIGHT	]	(inverse ']')
[RVS]	CTRL 9	R	(inverse 'R')
[OFF]	CTRL 0	■	(inverse line)

Instructions such as [DOWN * X] mean that the named key should be pressed X times.

Chapter One

Once Upon a Time...

... As Jar-Zel raced across the desert landscape, now cut with a lattice-work of bright red tracks, the Kharzog battlefleet moved silently overhead. From a distance they looked, to him, strangely familiar and not even remotely hostile – like floating pieces of that same gorg-board that made up the walls and roof of his own house. But those innocent-looking silhouettes were spraying the landscape below them with random bolts from their life-consuming ergonic cannon ...

GO WEST

... At the next intersection of tracks, Jar-Zel turned to his left and headed towards the nearest butte...

USE THE ZARN GUN

... Without breaking his stride, Jar-Zel pressed the belt stud that activated the zarn gun on his right wrist, then fired a beam of raw blue power towards one of the Kharzog ergon bolts. As the two energies met, the track of the ergon bolt became an incandescent pillar of fire, flashing back towards the Kharzog battleship that had launched it and consuming it in a single blinding explosion of energy...

GO NORTH

... Again Jar-Zel changed direction, turning the way he had intended to go. To his left the fallen ship, now half-buried in the sand, continued to blaze like the very sun...

SAVE GAME

... Now at last he could rest, safe in the knowledge that somewhere in a universe he could only imagine the sum knowledge of his world, his personality, his possessions, and the state of the battle itself had

been itemised, charted, and recorded – on an ordinary tape cassette.

Could you create an adventure like this, full of action, thrills, and high adventure? You may not think so now – but it isn't really as hard as it might seem. This book is about computer adventure games – what they are and how to prepare and program your own adventures for fun and/or profit. It is also quite different from any other book or article on the subject you will have read before.

O.K. – that's the sort of claim you'd expect an author to make about his book. But this time it happens to be true. In the following chapters you'll find out not only how to write an adventure program but also many of the tricks used in the best commercially available games. Your journey into the world of top class adventure writing starts here.

Back at the beginning

Once upon a time, many years ago in a far distant land, the wizards of the East and West entered into a fierce contest. At first it seemed certain that Crowther the Cunning and Woods the Wily – the wizards of the West – must win the struggle. For they brought forth from the dark tower of the Stanford Research Institute the wondrous computer adventure game named *Colossal Caves.*

But the wizards of the East were not so easily defeated. Before long Anderson the Artful, Blank the Boffin, Daniels the Devious and Lebling the Learned (whose home was among the misty halls of MIT) returned to the fray with *Dungeon*, an even more faithful reproduction of the game *Dungeons and Dragons* as invented at the dawn of time by the legendary Magi, Gygax and Arneson.

All of these events took place, believe it or not, a mere seven years ago, in America. At that time it seemed unlikely that the adventure type of game could be converted for use on any of the few microcomputers then available within the foreseeable future. But times change, and in the next few years several important events occurred which turned the improbable into accomplished fact. There was a drastic reduction in the cost of RAM chips, small (5¼ inch) disk drives became relatively commonplace, and Scott Adams began to write his series of micro-sized adventures.

Just how much progress has been made can be judged from the fact that a company called Level 9 has now released the original SRI Adventures, in unabridged form, for use on a 48K computer.

Indeed, in the case of *Colossal Caves* they have actually managed to *add* a further seventy rooms to the final section of the game!

Small is beautiful

So what is an adventure, and what separates the good games from their rivals? These are pretty big questions. The answers are the key to writing top class games.

Taking the first question first (and why not?), an adventure game can be the players' passport to the universe. One of the greatest gifts that adventure games bestow is the gift of freedom. The freedom to be anyone you want to be, at any time in history – past, present or future! The freedom to go wherever you want to go and to meet whoever you want to meet. The only limits to the world of the computer adventure game are those set by the ingenuity and imagination of the writer. A glance down the list of adventures already on sale shows how diverse that world has become. And the list continues to grow.

But one thing needs to be made quite clear right at the start – an adventure game is *not* an arcade game (despite some recent claims to the contrary). In an arcade game the player is required to control what happens on the screen using a set of keys, a joystick or a paddle. These games are basically about manual dexterity and reaction times. A true adventure game, on the other hand, calls for the player to take part in a story which is depicted either in text or in a mixture of text and graphics. Thus adventuring is mainly concerned with strategy, problem solving and mental rather than physical skills. An arcade game is over, in say, ten minutes or less (unless you happen to be extremely skilful). A good adventure takes hours, days or even weeks to solve, and the lower your skill the longer it will take to complete the game. Taking part in an adventure is, as one writer put it, like reading a book in which you are both the central character and co-author.

Unfortunately there are a couple of major hurdles facing the writer, or writers, of an adventure game, and it seems that quite a few authors are finding it difficult to jump those hurdles successfully. Take these comments from a couple of reviews published in the autumn of 1983:

> 'This is a text-only game and even the text formatting leaves a lot to be desired.'

> '(This game) might sound like a disaster movie, but in fact it's opening up a new field in the home micro market: disaster software.'

Maybe it's not surprising that so many poor-quality adventures are beginning to appear. Until quite recently most software houses have concentrated on producing home micro versions and variations of arcade machine originals. As a result the adventure market has, for quite a while, remained relatively untouched, especially in the UK. But now the 'adventure boom' is on – witness the rash of new programs, books, and even a specialist magazine. And just as the original arcade games have spawned a stream of second-rate look-alikes so, it seems, the core of high-quality adventure games will find itself surrounded, at least temporarily, by various ill-conceived, hastily-written and often poorly-programmed imitators.

Yet this very fact can be interpreted as good news of a sort. For I predict that even the dud adventures will help to create an appetite for top-rate games. I predict that in the long run adventure games will become the market leaders for computer software. And just as record 'singles' usually burn themselves out in a matter of months at most, while the best L.P.s go on selling year in and year out, so the best of the adventures will enjoy a lifetime that far exceeds that of the average arcade game. (How many people do you know who still play the original Space Invaders?)

The golden oldies

If that last prediction sounds like an adventure freak's dream, it's worth remembering that adventure gaming already has a 'Hall of Fame'. And among its growing number of inhabitants the pride of place must certainly go to the series of programs produced by the American company Infocom.

Infocom, like the game *Dungeon* referred to above, and like the LOGO computer language, has it roots in MIT – the Massachusetts Institute of Technology. Indeed, Blank and Lebling are two of the leading figures in the Infocom setup. Since it was founded, only three years ago, Infocom has produced a string of smash-hit adventures – *Deadline* and *Witness* (for the armchair detective), *Starcross* and *Planetfall* (for the earthbound astronaut), and *Suspended* (a 'hi-tech' game that is almost beyond classification).

But the greatest of all the Infocom creations, at least at the time of writing, has been the *Zork* trilogy: *The Great Underground Empire, The Wizard of Frobozz* and *The Dungeon Master.* Let's try to see why these games are so highly rated.

(1) Communication

A central feature of the Infocom programs is the facility for the player to 'talk' to other characters in the adventure. Readers who have encountered the British program *The Hobbit* (Melbourne House) may question the uniqueness of this function. After all, *The Hobbit* shares this facility and the Melbourne House language, known as Inglish (sic), bears more than a passing resemblance to Infocom's Interlogic.

In both languages the programmers have put a great deal of work into creating command 'parsing' routines which can cope with instructions from the player which look as much like standard English sentences as is possible within the limited RAM space of the average micro. Despite the complexity of this task (which will be explained in detail in Chapter 8), both companies have made significant progress. Thus Inglish can accept compound commands such as:

> SAY GANDALF "READ SIGN", GO N, UNLOCK DOOR, ENTER DOOR.

and Interlogic would cater for:

> TELL THE WIZARD TO READ THE SIGN. GO NORTH AND UNLOCK THE DOOR THEN ENTER.

The difference between these two sentences is quite obvious, though the practical differences may be much less noticeable. What both sets of writers are striving for is an end to the familiar two-word command format found in so many adventures even today. If we do want to draw comparisons then Interlogic appears to allow for a greater number of 'delimiters' – ways to define the end of each command module (as in normal speech) This gives a rather stronger impression that you really are talking to the computer rather than simply feeding it instructions.

(2) Commands

Anyone who has ever played an adventure game will have a good idea of the basic vocabulary available for use in commands:

GET
DROP
GO
READ
FIRE, etc.

But using such a basic set of instructions can often be more frustrating than helpful. Infocom claim, and there is no reason to doubt them, that Interlogic can access a vocabulary of around 600 (yes, six hundred) words!

I've already mentioned the difference between, say, Interlogic and Inglish – mostly it lies in the variety of words available for use by the player. The point is worth repeating, however, since it concerns one of the most critical factors in the success of any game – its 'believability'.

Since an adventure can be set at any time and anywhere, it's a fair bet that players will usually find themselves in worlds far beyond their own everyday experience (though not necessarily beyond their daydreams). This means that in the opening stages of an adventure, at least, the link between the player and the game will be somewhat fragile. In a good game, as with a well-written book, that link should develop and strengthen as the player becomes immersed in the action. This in turn requires that the game runs as smoothly as possible. Yet many commercial games still place that fragile link in jeopardy right from the start by having a very limited, and in some cases quite inconsistent, set of commands.

In Chapter 8 we'll be looking at ways to develop an extensive command vocabulary and upgrade command inputs to the level of normal speech.

(3) Room descriptions

Recent months have seen the appearance of a growing number of 'graphic' adventures. In some cases – *The Hobbit* and *Pirate Cove* (Scott Adams) for example – the graphics take the form of simple displays of the various locations. The more ambitious Stateside offerings – *Aztec, The Temple of Aphasai*, etc, – combine arcade-style graphics with an adventure-style plot. Yet most of the best adventures have remained true to the text-only originals. Not surprisingly, Infocom have managed to turn this apparent limitation into a positive virtue. According to one of their advertisements:

> 'You'll never see Infocom's graphics on any computer screen. Because there's never been a computer built by man that could

> handle the images we produce. And there never will be. We draw our graphics from the limitless imagery of your imagination ...?

This claim has been supported, to quite a large extent, by reviewers. The magazine *Softalk*, for one, described the text in *Zork III* as being 'far more graphic than any depiction yet achieved by an adventure with graphics'. The fact that all of the Infocom programs seem to have taken up permanent residence in the Softsel best-selling games list adds further credence to the company's far from modest self-appraisal.

With an introduction like this you might be forgiven for supposing that Infocom's room descriptions, etc., take up page after page of memory, and are beyond the scope of the average micro with less than 40K of free RAM space. Fortunately this is *not* true. And in any case, as Level 9 have recently demonstrated, there are ways and ways of fitting text into the space available.

In Chapter 2 I'll be showing you some of the main steps needed for the preparation of high-quality text displays. In Chapter 7 you'll find a BASIC version of the sort of 'text packer' that lies behind Level 9's success.

(4) The plot

The main requirements for the plot of a successful adventure game can be set out under three headings: (a) comprehensibility, (b) interest and (c) internal consistency. Of the three it could be argued that factor (b) is the most important. But only by a narrow margin. And failure to maintain a high standard in just one area is often enough to destroy the viability of the program as a whole. So let's look at these three factors in a bit more detail.

(a) Comprehensibility. Though you might not guess it from playing some of the games on the market at the moment, making a game *comprehensible* is certainly not the same thing as making it simple. To return to the Infocom series for a moment, not one of their games is exactly 'simple'. Indeed, *Suspended* is so complex that even though a game map and markers are supplied, one reviewer advised potential players not to 'worry about your score on the first few attempts – you'll have more than enough to cope with!'

To put it another way, what would you think of a book with only two or three pages, or a comic with just one picture on each page? If a game is too simple then it won't last long as far as playing time goes. And that can be quite a let-down if you've just paid out £20-£30. A

good plot is not one that is *easy* to see through, but one which makes sense when you take the time and trouble to get to grips with it.

Remember, an adventure takes place in its own little world. Getting to know that world – when it has been created with care – is half the fun of playing an adventure game.

(b) Interest. No game is going to attract much notice unless it both grabs the players' interest *and holds it*. Many are the stories of lone adventurers playing through the night, and even through a whole weekend, as they struggle towards their elusive goal. Although Infocom's earliest games followed the earlier 'sword and sorcery' format of the original *Dungeons and Dragons,* their later products – *Deadline* and *Suspended* – have explored new avenues in computer gaming. Such are the games which satisfy, and those are the games that sell through the best advertising available – word of mouth. In Chapter 2 I'll be dealing with the details of plotting and preparing an adventure game.

(c) Internal consistency. While it is impossible to lay down any hard and fast rules as to what should and should not appear in an adventure, it is essential that any game should have what amounts to a fixed set of rules. The most common failure in this area is the relationship of one 'room' to another on a game map. In second-rate games, one often finds that one can move from room A to room B, yet for no apparent reason it is impossible to return to room A. Worse still, one may move south from room A into room B and then find that moving north again takes you into room C (room A having disappeared, it seems, in a puff of smoke in the meantime!).

In some games this may be due, quite unforgivably, to poor programming. In other cases it occurs because the writers/programmers simply haven't bothered to prepare a decent map of their game before coding it. In both cases the result is sloppy and, for the player, incredibly frustrating. Such games deserve to fail when they reach the market place, and they usually do.

In Chapter 4 you'll find descriptions of the various ways of mapping out an adventure with their advantages and disadvantages. In Chapter 6 I'll be showing you how to store a map in the computer – in the form of *movement codes* – together with two methods of accessing the codes without using valuable array space.

(5) The problems

One of the central features of any adventure game is the range of problems set to trap, baffle and generally hoodwink the player.

Unfortunately this is one area where it is impossible to offer any very useful advice.

The main consideration is that the problems in a game should, by and large, relate directly to the plot of the adventure, and should be of a kind that could be solved by any reasonably intelligent person with a fair amount of general knowledge. This may sound rather narrow-minded, but I am looking at things from a professional point of view. It's true, of course, that adventure games were first developed by university students and staff, and that a large part of the market is probably still made up of people with a similar level of intelligence and education. But you don't have to be a genius to own a computer. And it certainly doesn't make much sense, in terms of potential profit, to aim at such a limited market. The best adventure games around at the moment certainly don't 'talk down' to the player, but neither do they demand that the player be a potential candidate for MENSA. Generally speaking the best problems require *careful* thought – and maybe a bit of 'lateral thinking' – rather than a store of specialised knowledge.

So, with these thoughts in mind, let's get down to business. Let's write an adventure...

Chapter Two
Plotting Your Adventure

The first step in preparing any adventure must be to set out a basic storyline, since everything else that you do will depend upon and relate to the plot of your adventure.

I should, perhaps, make it clear at this point that in this book the word *plot* is used to refer to the most basic outline of a story, while the word *storyline* is used to indicate an outline of all the important features of a story. Thus preparing a story goes through three stages:

Prepare the *plot*
Develop the *storyline* from the *plot*
Fill out the *storyline* to arrive at a finished *story*.

A successful plot will take three main factors into account:

(1) The need for players to 'believe' in the game
(2) The need for players to be satisfied with the roles they are given to play
(3) The need for players to feel that their efforts are being justly rewarded.

These are the guidelines that any adventure writers worth their salt will try to follow to the letter. Before we can follow the rules, however, we first have to find and develop a plot and storyline.

Finding a plot

In many respects there isn't a great deal of difference between creating a good adventure and writing a book or, to be more exact, a play or a film script. Obviously no game will take the same amount of writing as a complete book or script, but the basic approach will be pretty much the same. This may sound rather daunting if you've never tackled adventure writing before, but don't be put off. The

similarity is, as it turns out, a benefit rather than a disadvantage. For while professional adventure writers and programmers are, understandably, rather secretive about their work, there are plenty of books on the market that give step-by-step details of how to write short stories, a book, or a script. I won't pretend that there is room to cover this whole subject in just one chapter. What we can do is to take some of the basic 'tricks of the trade' and see how they can be applied to the art of adventure writing.

It has been said that the whole of literature is based on just six plots – three central ideas, each with a choice of two endings:

(1) The Love Story: A meets B – A gets B
A loses B
(2) The Search: A looks for B – A finds B
A doesn't find B
(3) Good Guys vs. Bad Guys: Good guys win
Bad guys win

Obviously no single story is quite that simple: the plots are open to a wide range of variations and can be strung together in many different combinations. And the action itself can be viewed from a variety of different positions. So the first thing to realise when you sit down to create a plot is this: the *originality* of your story, your adventure, doesn't depend on your ability to dream up a totally unique plot. It depends on your ability to *present* the story in a way that is fresh, interesting and exciting.

Actually we could go one step further and say that most people actually *prefer* stories that are both new and familiar. This may sound like a contradiction, but just think about the books you and your friends read. How many people do you know who read more than, say, three or four different kinds of book? One person will prefer science fiction or westerns, another usually picks detective stories or thrillers, someone else will choose mainly love stories or biographies. This much we know for a fact, and there's no reason to believe that the situation is any different when it comes to adventures.

Tip number 1: don't waste time trying to write a totally original game. You might just make it, but the odds are very much against you. And if you're trying to break into the commercial market it's more than possible that no one will want to buy a game that is totally unlike anything they've ever met before.

So if you're not aiming for something completely original in the plot line, where do you start? The answer can be found in advice

frequently given to first-time authors: don't try to tackle subjects you know nothing about – make use of what you already know. If you think back to some of the things I said in the first chapter you'll see how even the top writers have followed this advice.

Take the Infocom team as a typical example. They started out working on *Dungeon*, a direct development of the board game they already knew. When they turned professional their first *three* programs followed the same 'sword and sorcery' theme. Only then, when their talents had been firmly established, did they branch out into new territory.

Tip number 2: for your first attempt at writing an adventure take a theme you already know about, (preferably one that *interests* you since a high-quality program can take weeks or even months to complete!). Don't worry if your chosen theme is already covered in the list of commercial games; this only proves that there is already an established market waiting for you if your work is of a high enough standard.

The storyline

O.K. You've chosen a plot, now you have to construct a storyline around it - like setting the foundations and then building a house on top of them. The way you go about this task will depend on which method you find works best for you, and the steps that follow, both in this chapter and the next two, can be rearranged to a large extent to fit in with your own way of working.

One possible way of starting to build your adventure, and one that seems to be quite popular with many professional story writers, is to let the story write itself, so to speak.

To do this you need to start with just three items: a character, a starting place for the story and a possible ending. The reason why this approach is so popular is, I think, because it allows you to swap things around, or even alter the entire storyline, without having wasted a sizeable chunk of time in mapping out what *ought* to happen.

Where's your imagination?

Whether it's in relation to a school essay, a story, or an adventure, one thing that holds a lot of people back is the feeling that they don't have the imagination needed to produce something that anyone else would want to see. Yet being imaginative really isn't as difficult as

you might think. In fact it's usually the fear of not producing anything worthwhile that holds us back, rather than a genuine lack of ideas. Fortunately there is quite a simple solution to this problem, one that almost anyone can benefit from. It's called 'brainstorming'.

Despite its odd-sounding title this process doesn't require that you have a nervous breakdown. On the contrary, it has proved to be one of the most successful ways of generating original and interesting ideas that anyone has yet discovered. The theory behind it goes something like this.

Normally when we're asked for ideas, or even when we're just daydreaming, we tend to give each idea marks as though it were some kind of school exercise. In other words we decide whether an idea is 'good', practical, useful, etc. - or not - before the idea has really had time to develop. Obviously not all ideas are constructive, but if we give them a chance to develop - if we give our imagination a free rein - then even dud ideas can lead on to useful lines of thought.

So how do we put this to work? By taking note of *all* our ideas on a given subject and writing them down (this is a very important part of the process) - then weeding out the useless ideas *after* the brainstorming session is over. Thus, assuming that we've already come up with the bare details of a storyline, we sit down and think of everything that our main character *might* do in the course of the adventure, no matter how stupid, fantastic or irrelevant these actions may seem.

What will I need?

Having created an initial list of ideas - which we'll call **list 1** for the moment it might seem sensible to go straight on to sort out all the useless ideas and see what you're left with. In fact it is better to leave list 1 intact for the moment while you prepare **list 2.**

In this second list you should itemise all the characters, locations and objects you would need to include in your game if you used every idea in list 1. I say this for several reasons. In the first place it will help to give you some idea of just how much material you need to edit out of list 1. Secondly, the process of preparing list 2 may well help to suggest extra or alternative characters and objects, and may indicate alternative ways of dealing with the situations you have mapped out in list 1. What you're actually doing here is carrying the 'brainstorming' process one step further, in a more immediately practical direction, without closing off your options. Which brings us to step three...

Back to reality

We might describe the two lists we've prepared so far as follows:

List 1 = 'the possible' (however incredible it may be)

List 2 = 'the practical' – what actually has to go into the computer

With both lists in front of you it's time to decide what you're going to keep and what is to be discarded. To make these decisions you will need to bear two factors in mind.

First with regard to list 2, how much of the inventory will actually fit into the RAM space you have available? A good deal of the information in this book is concerned with ways of packing as much into an adventure as possible in quite a limited amount of RAM. So you should find, unless list 2 is very long indeed, that you don't need to do too much editing at this stage.

Secondly, going back to list 1, after all the totally outlandish ideas have been rejected how much of what is left can you actually fit into your program? In other words, how good a programmer are you? This will depend as much as anything on your experience of programming in general, and is very much a case of 'practice makes perfect'.

Tip number 3: Know your limitations as a programmer. There are few things as frustrating as getting halfway through coding a program and finding that you don't know how to deal with a key sequence. Aim to improve with each program, but don't set yourself goals that you have no real hope of achieving.

Blocking it out

Once you've completed that last stage in your preparations, you should have two lists of fairly manageable size on which to base your adventure. At the risk of seeming repetitive I would emphasise that the material you have left should be regarded as 'shortlisted' material rather than something which is fixed and unchangeable. Creating an adventure, despite its partly scientific element (the actual programming), is essentially an *artistic* enterprise. And part of the pleasure of any artistic undertaking is that until your work is finally completed even you, the artist, cannot be entirely sure what you're going to end up with. Writing an adventure game is itself a kind of adventure, and since it will certainly involve a fair amount of hard work there's no earthly reason why you – the writer/programmer – shouldn't also get as much fun out of it as you can.

At this stage of your preparations there is still plenty of room for experiment and changes of mind. But be careful how many changes you make! It can be tempting, when a really good idea comes along, to rewrite your previous storyline to make use of the fresh material. Experienced writers, however, soon learn to be more economical with their ideas. For it's just possible that the ideas that come along while you're creating one story are strong enough, if properly developed, to provide the basis for another complete story. To put it another way – if you use up all your best ideas in your first adventure then what will you use for the next one?

Talking of stories, it's now time for us to put one down on paper in such a way that we can start to build a program around it.

The storyboard

The actual process of writing a story is a pretty personal thing, and it would be impossible to set out one method of tackling this job that would suit everyone. This section, therefore, is, definitely not about how you *should* approach the task, but rather how you *might* approach it.

It really is true that writing a computer adventure game is a lot like preparing a film script (which is where the storyboard technique was developed). For where a film consists of a number of 'set pieces' or scenes in which the central story evolves towards its climax, so an adventure moves through a series of fixed locations as the player tries to gain the maximum score or achieve the goal which the writer has set.

I don't want to push this comparison too hard, but it does raise at least one issue that every adventure writer needs to think about – the importance of imagery, transferring a picture from the scriptwriter's pad to the audience's mind.

If it's true that 'one picture is worth a thousand words' then obviously film-makers have a terrific advantage over the adventure writer in this area. Which is why Infocom, for example, make such a big thing out of the quality of the text displays in their programs. But this isn't to say that good writers will automatically produce good text displays for an adventure game. Nor can we assume that the addition of graphics will automatically improve the quality of a program. (Indeed it has been said that graphics can actually spoil the effect of a game unless they are of a particularly high standard. For a more detailed discussion on this subject see Chapter 10.)

And that's where the storyboard comes in.

If you've ever seen one of the TV programmes which show how films are prepared then it is quite probable you already know what a storyboard is. For those readers who don't know what I'm talking about, a storyboard is a kind of comic-strip version of the film script; the director and cameraman use it to work out what will happen in each scene, how it will be lit, what camera angles will work best, etc., etc. In a sense, list 2 described above is a text version of a storyboard in note form. What we need to do now is to turn the notes into individual, itemised scenes. What follows is an illustration of how the storyboard for one scene (one text screen) in an adventure was developed from the original listings to the point where it was ready to form part of a program:

List 1

Hero meets rogue cybernaut which is carrying the only key to the lab. door. The door is the only way out of the room and the key is the only way of opening the door. The room is on fire. The cybernaut is heatproof, the key isn't!

List 2

Characters: Hero, cybernaut
Objects: A key (a magnetic card?)
Problem: Get key to open door before cybernaut kills you or you are burnt to death.
(Need extra object?)

Reading these notes over, the idea looked O.K., but I remembered that the term 'cybernaut' came from *The Avengers*. I decided to make do with an ordinary robot.

Version 1: Screen 24. Laboratory.
You are locked in the laboratory with a highly aggressive robot. As the robot moves towards you it knocks over a bench bearing several bottles of fluid. When the bottles hit the floor they break. The fluids mix and burst into flame. The robot is obviously heatproof as it is still coming towards you through the flames. The robot has the only key to the door.

What now?

Hero, robot, magnetic card, (extra item?)

So far, so good. But the text doesn't look right yet, and the robot has no character.

Version 2: Screen 24. Laboratory.
The door slides open and you step into the Professor's laboratory. The door closes automatically behind you.

The only key to the door is a magnetic card lying about halfway between you and Igor the Professor's psychopathic android.

Igor moves towards you but his arm catches a row of bottles which crash to the floor. As the chemicals mix they burst into flame.

What now?

Hero, Igor, magnetic card, (extra item?)

In case you haven't guessed, I don't know how to beat Igor yet! Still, the text looks better and I like Igor. On the other hand I don't see why I should make the presence of the magnetic key so obvious. I could make the player use the LOOK command before they can find it.

Version 3: Screen 24, Laboratory
With a quiet hiss the steel door slides open and you step into the Professor's laboratory. The door closes automatically behind you and you are trapped with Igor - the Professor's psychopathic android.

As Igor moves through the cluttered equipment towards the table which stands between you his arm catches a row of bottles, sending them crashing to the floor. The chemicals mix and ignite in a ball of flame.

What now?

Hero, Igor, magnetic card, (extra item?)

At this point the text has reached its final state, and I decided to move on to the second part of the storyboarding technique - planning possible moves and their consequences.

In the player's position I might well try to move out of harm's way. But this won't be possible within the laboratory. I could make the response to *any* movement (i.e. GO NORTH GO SOUTH, etc.) 'Bad luck - you're dead', but it's a little soulless. In the end I decided on three different responses:

GO NORTH (i.e. back to the door)
The door won't open - and Igor's getting closer!'

GO EAST or GO SOUTH
'Throwing the table out of his way, Igor seizes you by the neck and squeezes the life out of you.'
(Go to the end of game sequence)

GO WEST
'Too bad, you escape Igor's clutches only to be roasted alive.'
(Go to the end of game sequence)

So much for the possible movements. What about the possible actions? (At this point I stopped and planned a way in which Igor could be beaten – the following orders and responses mirror this new development.)

OPEN THE DOOR
'You can't – not yet!'

LOOK AT THE TABLE
'There is a striped plastic card on the table. It is the key to the lab. door.'

GET THE KEY or GET THE CARD
'Igor moves faster than you. Reaching across the table he grabs you by the neck and throttles the life out of you.'
(Go to end of game sequence)

STOP/KILL/DESTROY IGOR
'How are you going to do that?'

ANY WRONG ANSWER
'Too bad – Igor was so unimpressed that he rushed forward and broke your spine in a fatal bear hug.'
(Go to end of game sequence)

WITH THE REMOTE CONTROL
'Well, well – Igor had an achilles heel after all. The TV control has blown his circuits and he is unable to move a single microchip.'
'What now?'

I won't go into all the possible variations on this theme, but I'm sure you get the general idea – stop Igor with the remote control unit (which I now have to introduce in an earlier scene), grab the key and leave the laboratory. This is the *only* acceptable solution and all other lines of action result in the player being killed.

This part of the preparation of a story can be quite hard, given

that you have to try to cover every possibility, and inevitably gets boring after a while. It's worth remembering, however, that you may not come back to the storyboard until you start coding your program. And it's a lot easier to stamp out the bugs when everything is written down in plain English than it will be if you have to start adding lines, renumbering, etc. This really can be a case of 'a stitch in time saves nine'.

Chapter Three

Who Goes There?

It isn't everyone, of course, who can just think of an idea and then weave a complete story around it. And there's no reason why they should, any more than everyone *should* like Raspberry Ripple (or any other ice cream, if it comes to that). A process of trial and error may well lead you to decide that you are better at creating characters first, and then playing around to see how they might react to each other. This is certainly how many well-known writers work, and when 'the plan comes together' the results can be quite fascinating. But just how do you go about creating characters?

There are few experiences more daunting for a writer than sitting in front of a blank sheet of paper and wondering how on earth he is going to fill it up. It can seem as though you'll never have another original thought in your life. And the longer you sit there, the worse you feel. And the solution? Don't get stuck with a blank sheet of paper!

That may sound rather facetious but I mean it quite seriously. One of the easiest ways to create a cast of characters for an adventure is to take a ready-made list and then alter it to suit your own requirements.

In the last chapter I said that first-time adventure writers would do well to stick to the kind of plot they already know and like. The same thing applies to creating characters. If you like horror stories, then start your list with a vampire, a werewolf, a couple of monsters and so on. If you prefer detective stories why not begin with Sherlock Holmes, Dr Watson, Inspector Lestrade and Professor Moriarty? At this stage of your preparations you will be creating a *framework* for your story, not the finished article, and if a little plagiarism helps to generate ideas, then do it. You'll find that getting something written at the top op the page will almost certainly help to give you the ideas you need to fill the rest of it. And once the ideas begin to flow you can move on to the next stage - organising the

characters in a way that will provide the basis for an interesting game.

Fixed vs. progressive

One of the most fascinating features of the *Dungeons and Dragons* (or *Trolls and Caverns*) board game is the chance it offers players to develop their personalities and abilities as the game evolves (as long as they don't get killed, of course). Thus a character who starts off as a down-at-heel sneak thief can gather treasure, learn spells, gain skill in the use of weapons and so on until, after a time, he begins to take on a whole new appearance. It's hardly surprising, therefore, that many players become attached to a favourite character whom they 'freeze' between games so that he, or she, can continue to grow and develop over a period of months or even years. (It's not unknown for such characters to be resurrected if fate deals an unkind hand to their careers!)

If we apply this idea to the field of computer adventure games we immediately come up against one of the great unsettled questions of the day: should a player's (fantasy) personality be dictated by the player, the writer, or the computer (i.e., by controlled random selection)?

In many games now on the market this problem simply doesn't arise, since the player's character remains virtually static from start to finish, depending entirely upon the player's skill and judgement to complete his task. Although it is certainly a lot easier to write and encode this kind of adventure program, the situation has several limitations which are worth considering.

In the first place, as the last paragraph suggests, a character which is totally 'fixed' tends to appear as little more than a cardboard cutout. For someone to go through the trials and tribulations of an interesting game without registering any positive response – fear, caution, etc. – detracts from their credibility, and it is very difficult for the player to feel that such a character is really *involved* in what is going on. The player will not be drawn into the game by his game character; instead he will be more likely to feel that there is a tangible gap between himself and the action on screen. In other words he will be aware that he is indeed just 'playing a game'.

The second objection to the 'fixed' character is that the game itself will lack the important sense of progress that is possible in the original board game. If this element of character growth is missing

then the attraction of a game rests to a very large extent on whether the player can work up any enthusiasm for the task he has been set. Just scoring points, or completing a certain percentage of the whole game, is O.K. in a fast-moving arcade game, but it offers very limited satisfaction after hours or days of effort over a relatively slow-moving adventure.

I'm not saying that it is possible, even now, to produce an entirely accurate representation of *Dungeons and Dragons* on a computer. Nevertheless, the most successful computer versions *have* managed to transpose the main features of the original game – which is almost certainly *why* they are so successful!

The third and last drawback that I want to deal with here is concerned, to a large extent, with commercial games rather than those which are written simply for fun or practice. The problem is the value of the game to the player. If a game is essentially 'static' – if its only purpose is to score points and/or carry out a pre-set task – then once the player has successfully completed the game it is, so to speak, dead! Unless you happen to like action replays – or have a truly appalling memory – there is no point in playing the game again. All you can do is go out and buy another game, which is fine for the games writers, but not so good for the players.

It may be that there aren't enough good games on the market at the moment for players to expect anything better. But it is clear, from the latest reviews at the time of writing, that most of the best new ideas in adventure programming – the use of compound instructions in standard English, for example – are being incorporated into each new generation, so the overall quality is bound to rise considerably, even within the next year or two. Whether you plan to write your games for pleasure or for profit there isn't much point in learning a style of presentation which is already becoming out-of-date.

So what is the alternative? Actually there are several choices, and I'll be discussing them, together with examples of how to program each one, later in this chapter. But first I want to look at the actual process of creating or adapting the characters for an adventure.

Zero population growth?

If you've ever come across the game *Wizardry*, or read one of the numerous reviews, you'll know that it features the relatively unusual ability to handle up to six player-controlled characters at a time.

This is no easy task, and it is only possible because *Wizardry* - like most of the top adventures - is written entirely in machine code. Its great advantage is that unlike most other games the player can afford to use several characters as scouts, decoys, etc, and even lose them altogether, without running the danger of being sent back to start a fresh game. (Obviously there is a limit to the number of risks you can run even with this advantage, but six lives have to be a lot better, from the player's point of view, than one.)

In a later chapter I will be showing you how to move minor characters in an adventure around at the same time as the player is moving. For the time being, however, I will work on the assumption that you will have only one *player-controlled* character.

Hail the conquering hero

It is an essential feature of any adventure that both the storyline *and* the characters be consistent within their own little world. One could, for example, have a hairy-chested, axe-wielding barbarian as a character in a space adventure if that really took your fancy. But a seven-foot, bright purple, blood-crazed alien with a laser spear would surely fit the part just as well (especially if you taught it English), and it would probably fit much more realistically into the main storyline. So when you begin to create your central character - the one which will be controlled by the player - there are at least two important factors to be considered.

Firstly, try to make your hero/heroine a little bit out of the ordinary. Remember part of the fun of adventuring lies in taking the players away from their everyday world of the classroom, the kitchen sink, the office or the factory floor. The stronger the link that you create between players and fantasy characters the greater will be their enjoyment. Restarting a game should ideally be more like rejoining an old friend than simply switching on the computer when you have an hour or so to spare.

Tip number 4: think player. Your main enjoyment will come from creating a game that intrigues and fascinates your friends, and, with a bit of luck, one of the big software houses. The player's fun comes from games which offer genuine thrills, surprises and involvement. So always try to keep in mind a picture of the sort of person you think might enjoy your game. Try to imagine their response to the character you have given them, and the things that happen during

the game. This may not sound an easy task, but if you are successful you will find that it can help you in more ways than one.

Let's return, then, to our would-be hero or heroine. How do we give them that little 'extra something' that will lift them out of the ordinary?

Again this is something that is easier to do than to describe, so let's take a typical character from the world of fiction – the spy – and see how he has been depicted over the years.

If we go right back to the turn of the century we find that spies played a very small, and usually unpleasant, role in the novels of that time. The British public still regarded spying as a rather loathsome occupation best left to villains and foreigners. So in the most famous spy story of the time, Erskine Childers' *Riddle of the Sands*, the hero is an innocent civilian who stumbles on a dastardly plan by the German navy quite by accident. Yet even in this watered-down form the ideas behind the story were regarded by many people as being thoroughly unsportsmanlike!

This attitude changed quite radically in the period immediately following World War I, and as a result the gentleman spy began to appear on the bookstalls of the day. Characters like Bulldog Drummond and, much later, Dick Barton – Special Agent. This new breed of spies, or 'spy catchers' (a very subtle distinction), were broad of chest, always kind to women, children and animals, and often none too bright, to judge by their mistakes.

But times change, and the next character type to appear was James Bond and his imitators. Though still a gentleman (of sorts) the Bond-like breed were more in tune with the permissive society and the growth of high technology. They relied less on brute strength and more on seduction, and whatever weird and wonderful new invention 'Q', or one of his fellow-workers, could produce.

Coming right up to date the pattern has changed again. The most recent figures at centre stage are the 'mackintosh brigade': men like Callan, Harry Palmer and George Smiley. Ordinary people doing an unpleasant but necessary job in an unpleasant world, often under the orders of men who would make good candidates for the KGB.

Looking back, each of these character types may seem a little old-fashioned now. But each was a true original, and a best-seller, when it made its first public appearance. As I said in the last chapter, total originality is nearly impossible, but a few small changes to an established character can give the *appearance* of creating something entirely new. And that's what counts.

Of men and supermen

The second point I want to make about creating a hero is this: resist the urge to create a super-hero. At the risk of seeming totally obscure I believe that the games that last are the games that last. In other words, a good adventure game is one that takes a long time (within reason) to complete. Indeed, some leading firms make it a central feature of their advertising that it will take weeks or even months to work right through one of their adventures. And when you think how much the top games cost, this isn't a bad selling point.

But how long can an adventure last if the hero is so powerful, intelligent and lucky that he can overcome all obstacles at the blink of an eye? In the best games, the player usually has a less than 50-50 chance of completing the adventure at the first try. The addictive quality of these games comes from the player's ability to turn the odds gradually in his favour with each attempt.

Tip number 5: be very careful how you stack the odds. Make them too large *against* the player and he or she will soon become frustrated and disheartened. Stack them too heavily in the player's favour, on the other hand, and you remove the challenge, the excitement and the sense of achievement that are essential to the game's success.

And the monster came too...

Having decided on a personality for your star character you will now need to provide 'a full supporting cast'. If you have already drawn up a shortlist by one of the methods described earlier then this shouldn't present too much of a problem. It will be useful, however, if you can decide right away whether they will be leading characters or merely extras. The uses of second and third level characters differ in several ways and can, therefore, be defined quite clearly.

Leading, or *second level*, characters often appear on several occasions over the course of an adventure, mainly because their relationship with the hero will play a major part in deciding the outcome of the game.

Where such a character is one of the 'good guys' he/she will possess special information, a useful object, or unusual powers which can be of help to the hero. If the character is one of the 'bad guys', on the other hand, he will be intent on hampering the hero's efforts. Possibly with fatal consequences! Unfortunately for the player these

characters are seldom identified in advance, so the hero will have to use his own judgement in sorting the sheep from the wolves.

To illustrate the difference between second and third level characters let's suppose that you've invented a little old man but haven't yet decided how he will fit into the game.

If your little old man is a *third level* character then he will almost certainly exist at only one location, though it helps add to the player's confusion if one or two minor characters appear more than once! Should the player meet this character then, depending on his role, the old man may fulfil one of three basic functions:

(1) If he is a 'good' character then he may, if dealt with appropriately, offer the hero a piece of mildly useful information – possibly in the form of a riddle – or provide him with an object that will be useful (though not too useful) elsewhere in the game. Whatever he offers it should add to the *interest* of the game rather than playing a decisive part in its result.

(2) Since the little old man is only a minor character he may be in the game entirely for the purpose of giving the player something to do. Thus he might set the hero a problem which is interesting in itself, but of no relevance to the game whatever.

(3) Even minor characters may be set against the player, though the results of their actions should prove a hindrance, rather than being fatal. An evil third level character might misdirect the hero by giving him wrong information, or by questioning the value of an earlier clue or object which is of real value. However, since the character should not have undue influence the player must be given a reasonable chance of ignoring the little old man if he so wishes.

If, on the other hand, the little old man is a *second* level character his actions, and the hero's treatment of him, should have a significant effect on the progress of the game. In this case the little old man may be a powerful ally or foe in disguise, thus giving the hero's behaviour towards him far greater importance. A second level character will never offer totally useless information or advice. But he may retain his most important knowledge until the hero earns it or asks for it in the right way. Thus he might pose the hero a problem *and*, if treated correctly, provide the answer. In short, third level characters will usually rate the same degree of importance as any other minor task, trap, object or problem facing the adventurer.

Leading characters will be nearly as important as the hero himself. This may mean taking a bit of trouble over their creation, but this

will be repaid by their contribution to the credibility of the game. The more real these second level characters are made to appear, the more satisfying the adventure will be for the players.

Curtain up

I said earlier that I would give details of how to program a character. In fact I have included four of these 'character generators', each with a different approach to the question of who decides on a character's personality. At the end of the chapter you will find a further routine which recalls the character details and presents them on screen for easy reference.

Of course you don't *have* to include a character generator at all. But even in this case, for reasons which will be made clear in Chapter 5, I would suggest that you at least give your hero a strength rating.

And now – to the keyboard!

Program 3.1: The built-in character

In this first, rather short program the hero's character is fixed by the writer at the start of the game but can be altered during the adventure if the hero gains wealth, gets injured or whatever. The advantages of this method are (a) it is easy to program, (b) the writer knows exactly what shape the hero is in at the start of the game and can balance the problems accordingly, and (c) the routine itself takes up very little RAM space.

The disadvantages are (a) the players may not agree that the character they have been given has a fairly balanced personality, and (b) the fixed nature of the hero makes this a 'one shot' game. Once the adventure has been completed there are no alternatives to experiment with.

```
]LIST

1  REM *****  FIXED CHARACTER  *****
2 :
3 :
8  REM *** SET SCREEN COLOURS
9 :
10  POKE 53280,15: POKE 53281,15
17 :
```

```
18  REM *** GET NAME FOR PLAYER
19 :
20  PRINT "[CLEAR][DOWN * 8][PLEASE EN
     TER A NAME FOR YOUR CHARACTER"
30  INPUT "AND PRESS <<RETURN>> "NA$
97 :
98  REM *** SET RANDOM CHARACTER
99 :
100 CH =  INT ( RND (1) * 4) + 1
107 :
108  REM *** AND READ RATINGS
109 :
110  FOR X = 1 TO 4
120  IF X <  > CH THEN  READ A$,A,B,C
     ,D,E,F,G,H
130  IF X = CH THEN  READ CT$,CT(1),C
     T(2),CT(3),CT(4),CT(5),CT(6),CT(
     7),CT(8)
140  NEXT
197 :
198  REM *** DIRECT STORAGE
199 :
200  FOR X = 1 TO 8
210  POKE 700 + X,CT(X)
220  NEXT
227 :
228  REM *** INCLUDING NAME
229 :
230  POKE 712, LEN (NA$)
240  FOR X = 1 TO  LEN (NA$)
250  POKE 712 + X, ASC ( MID$ (NA$,X,
     1))
260  NEXT
270  END
297 :
298  REM *** CHARACTEF DATA
299 :
300  DATA WARRIOR,10,10,5,10,6,200,2,
     4
310  DATA WIZARD,7,7,10,10,6,150,10,1
     0
320  DATA THIEF,6,6,8,8,5,140,3,8
330  DATA DELVER,6,6,6,5,5,160,1,4
```

Program 3.1

Line-by-line analysis

Line 10: Sets the colours for the screen and border. The POKEd values may be altered to get whichever combination you prefer.

Lines 20-30: A simple display asks for the player's (fantasy) name which is then stored as NA$. This must be done in *two* statements or the INPUT command may not work.

Lines 100-140: In this version of the program a choice of four 'fixed' characters is selected randomly. For a truly fixed character delete lines 100-120 *and* line 140 and remove the first part of line 130 (that is, everything *before* READ) so that it becomes a simple READ instruction. The purpose of line 120 is to clear any unused DATA by assigning it to a set of 'dummy' variables.

Lines 200-270: POKE the values of the CT() array plus the player's name into a free area of RAM that is *not* affected by NEW, CLEAR or RUN. This allows the character generator to RUN separately from the main program without losing the stored information. Line 250 splits NA$ into individual characters, starting from the left, so that their ASCII codes can be stored in memory.

Lines 300-330: This is the DATA for the loop in lines 110-140. If you have made the modification suggested above then you will only need one line of DATA here.

Program 3.2: The made-to-measure rack

Our second program is a direct offshoot of Program 3.1. This time the player has the option of choosing one character type from a preset list. As in Program 3.1 the ratings for each character are also fixed by the writer. The advantages here are the same as before. An added bonus comes from the fact that, using the program below, the player would have four different guises in which to tackle the adventure instead of only one. This doesn't mean that he or she will necessarily get four times as much entertainment, but it does offer a significant improvement.

The disadvantages of this second program are mainly to do with the actual coding of the routine. In the first place it is more complicated. And secondly it takes up more RAM space; just over twice as much.

```
]LIST

1  REM *****  FIXED WITH OPTIONS  ***

2 :
3 :
8  REM *** SET UP TITLE ARRAY
9 :
10  DIM CT$(4)
20  FOR X = 1 TO 4
30  READ CT$(X)
40  NEXT
97 :
98  REM *** GET PLAYER'S CHOICES
99 :
100  PRINT "[CLEAR][DOWN * 8]PLEASE E
     NTER A NAME FOR YOUR CHARACTER"
110  INPUT "AND PRESS <<RETURN>> ";NA
     $
120 NA$ = "[RVS]" + NA$ + "[OFF]"
130  PRINT "[CLEAR][DOWN * 4]"NA$" PL
     EASE SELECT YOUR"
140  PRINT "CHARACTER TYPE FROM THE L
     IST BELOW:"
150  FOR X = 1 TO 4
160  PRINT "[DOWN]("X") "CT$(X)
170  NEXT
180  INPUT "[DOWN * 2]ENTER NUMBER OF
      CHARACTER TYPE NOW ";CN$
190 CN =  VAL ( LEFT$ (CN$,1)): IF CN
      < 1 OR CN > 4 THEN TE = 1
200  IF TE = 1 THEN  PRINT "[DOWN][RV
     S]YOU MUST CHOOSE A NUMBER BETWE
     EN 1 AND 4[OFF]"
210  IF TE = 1 THEN  PRINT "[HOME][DO
     WN * 13]":TE = 0: GOTO 180
297 :
298  REM *** GET RATINGS FOR CHAR' TY
     PE
299 :
300  DIM CT(8)
310  FOR X = 1 TO 4
320  FOR Y = 1 TO 8
330  READ CT(Y)
340  NEXT Y
```

```
350  IF CN < 4 AND X = CN THEN  GOSUB
     600:X = 4
360  NEXT X
370 CT$ = CT$(CN)
380  END
497 :
498  REM *** CHAR' TYPES AND RATINGS
     DATA
499 :
500  DATA WARRIOR,WIZARD,THIEF,DELVER

510  DATA 10,10,5,10,6,200,2,4: REM W
     ARRIOR
520  DATA 7,7,10,10,6,150,10,10: REM
     WIZARD
530  DATA 6,6,8,8,5,140,3,8: REM THIE
     F
540  DATA 6,6,6,5,5,160,1,4: REM DELV
     ER
599 :
600  FOR CL = X + 1 TO 4
610  READ A,B,C,D,E,F,G,H
620  NEXT CL
630  RETURN
```

Program 3.2

Line-by-line analysis

Line 10: This line sets up, or 'initialises' a 4 × 1 array to hold the four character types named in line 500.

Note: Strictly speaking the C64 does not require an array to be DIMensioned if it is this small, as it has a 'default' minimum size – 11 elements – which it allocates to any array which is accessed before it has been DIMmed. However, once an array has been used in this way the computer deals with it as if it had been correctly DIMmed. In other words, it cannot be re-DIMmed at a later date (check your manual for details). It is better to set *all* arrays before use, therefore, rather than crash the program with an incorrect array call. If you aren't familiar with arrays and their uses see Chapter 5 for more details.

Lines 20-40: This is a simple FOR...NEXT loop to store the names in line 500 in the array CT$().

Lines 100-120: These lines clear the text screen and collect a name for the player's character which will be stored as NA$ (in *reversed* characters).

Lines 130-140: The printout is the heading for a simple screen display, or 'menu', which allows the player to choose a character type from a preset list.

Lines 150-170: Another FOR...NEXT loop, this time to print out the four character types in the form:

(1) WARRIOR
(2) WIZARD
(3) THIEF
(4) DELVER

Lines 180-210: These lines finish off the menu and include an 'error trap' so that only the legal numbers – 1, 2, 3 and 4 – will be accepted by the computer. By inputting a *number* as a string and converting it (line 190) you avoid the ?REDO FROM START error message which comes up if a letter key is hit while a program is waiting for numerical input.

Line 300: See line 10

Lines 310-370: Here we have a loop within a loop, that is to say 'nested' loops. The outer loop, starting in line 310, will collect from one to four sets of character ratings. It is short-circuited in line 350 if necessary by resetting X to the last value in line 310 (in this instance 4). In this way we avoid jumping out of the loop before it is complete – something which is generally regarded as 'a bad habit'. See Chapter 9 for more details.

The Y loop, starting in line 320, collects eight character ratings from the DATA in lines 510-540 and stores them in the array CT(), ending when the required set is in the array.

Line 370: Although we are already using the variable name CT$() for an array some computers allow the first part of the name (i.e. everything before the first bracket) to be used separately. Here CT$ is used to store the name of the character type and the array CT$() can be re-used for something else to save space.

Line 500: Initial storage area for the information to go into the CT$() array.

Lines 510-540: Initial storage space for the character ratings for the CT() array. (In Chapter 6 you'll find a method of storing numerical data more efficiently.)

Lines 600-630: Because DATA is always read in the order that it appears in the program you can't skip over a portion that has not

been used. This subroutine clears any excess DATA from lines 510-540 so that future READ commands will not collect the wrong information.

Figures 3.1 and 3.2 show the actual screen displays generated by Program 3.2. The word BEOWULF, in Fig. 3.1, was of course input from the keyboard. In this second program I have left the numerical information in an array - CT(). It could, however, be stored directly in RAM using the routine in lines 100-120 in Program 3.1.

```
PLEASE ENTER A NAME FOR YOUR CHARACTER
AND PRESS <<RETURN>>  BEOWULF
```

Fig. 3.1. Screen display from Program 3.2, line 210.

```
BEOWULF PLEASE SELECT YOUR
CHARACTER TYPE FROM THE LIST BELOW:

(1) WARRIOR

(2) WIZARD

(3) THIEF

(4) DELVER

ENTER NUMBER OF CHARACTER TYPE NOW 2
```

Fig. 3.2. Screen display from Program 3.2, lines 220-230.

Program 3.3: User-controlled character type and ratings

The third program of this series, though it starts out looking a lot like Program 3.2, offers an entirely different approach to character creation. Here the player not only selects the character *type* but also determines the number of rating points to be given to each characteristic or 'character quality'. The writer controls this process only to the extent that he determines the number and names of the character types, and the upper and lower limits of the number of points that may be assigned to each quality. For the sake of consistency alone, the writer also controls the height and weight of each character type.

The biggest advantage of this system is that it extends the range of the game enormously while taking up comparatively little extra

RAM space. At last the *player's* imagination is set to work immediately the game starts. Will he be a Warrior with brains as well as brawn , or a weakling Delver who trusts to luck rather than skill? The biggest disadvantage lies in the high proportion of space-eating text. This should be more than compensated for, however, if you use the 'text crunching' routine described in Chapter 7.

```
]LIST

1  REM *****  USER-SET CHAR'S  *****
2 :
3 :
8  REM *** DIM STORAGE
9 :
10  DIM CT$(4),CT(8),H1(4),H2(4)
97 :
98  REM *** SET UP ARRAYS
99 :
100  FOR X = 1 TO 4
110  READ CT$(X),H1(X),H2(X)
120  NEXT
197 :
198  REM *** ASK FOR NAME
199 :
200  PRINT "[CLEAR][DOWN * 8]"
210  PRINT "PLEASE ENTER A NAME FOR Y
     OUR CHARACTER"
220  INPUT "AND PRESS <<RETURN>> ";NA
     $
230 NA$   "[RVS]" + NA$ + "[OFF]
237 :
238  REM *** AND CHAR' TYPE
239 :
240  PRINT "[CLEAR][DOWN * 4]"NA$" PL
     EASE SELECT YOUR"
250  PRINT "[DOWN]CHARACTER TYPE FROM
      THE LIST BELOW:"
260  FOR X = 1 TO 4
270  PRINT "[DOWN]("X") "CT$(X)
280  NEXT
290  PRINT "[DOWN * 2]ENTER NUMBER OF
      CHARACTER TYPE NOW ";
300 TE = 0: GET CN$: IF CN$ = "" THEN
     300
307 :
```

```
308  REM *** VALIDATE INPUT
309 :
310 CN =  VAL (CN$): PRINT CN: IF CN <
     1 OR CN > 4 THEN TE = 1
320  IF TE = 1 THEN  PRINT "[DOWN][RV
     S]YOU MUST CHOOSE A NUMBER BETWE
     EN 1 AND 4[OFF][UP * 6]": GOTO 2
     90
397 :
398  REM *** ASK FOR CHAR' POINTS
399 :
400  PRINT "[CLEAR][DOWN * 3]"NA$" YO
     U HAVE 48 RATING"
410  PRINT "POINTS TO BE DIVIDED BETW
     EEN 6 CHARACTER";
420  PRINT "QUALITIES (USE WHOLE NUMB
     ERS ONLY!)"
430  PRINT "[DOWN]EACH QUALITY MUST B
     E GIVEN AT LEAST 1  ";
440  PRINT "POINT.  NO QUALITY MAY BE
       GIVEN MORE    THAN 12 POINTS!"
450  PRINT "[DOWN * 2](1) STRENGTH"; TAB(
     20);"(4) SKILL"
460  PRINT "[DOWN](2) HEALTH"; TAB( 2
     0);"(5) WEALTH UNITS/10"
470  PRINT "[DOWN](3) INTELLIGENCE"; TAB(
     20);"(6) LUCK"
480  PRINT "[DOWN * 2]"
490  FOR X = 1 TO 6
500  PRINT "ENTER POINTS FOR"X;: INPUT
     "AND PRESS RETURN";Z$:Z =  VAL Z
     $
510  IF (Z < 1 OR Z > 12) OR Z <  >  INT
     (Z) THEN  PRINT "[DOWN][RVS]ILLE
     GAL ENTRY!!![OFF][UP * 3]": GOTO
     500
520  IF CH + Z + 6 - X > 48 THEN  PRINT
     "[DOWN][RVS]YOU ONLY HAVE"(48 -
     CH)"POINTS LEFT![OFF][UP * 3]": GOTO
     500
530 CH = CH + Z:CT(X) = Z: PRINT "[DO
     WN]      (32 SPACES)
        [UP * 3]"
540  NEXT X
596 :
```

```
597  REM *** MODIFY CT() ARRAY
598  REM       (SEE LINE NOTES)
599 :
600 CT(7) = CT(5):CT(5) = H1(CN)
610 CT(8) = CT(6):CT(6) = H2(CN)
620 CT$ = CT$(CN)
630  END
9996 :
9997  REM *** CHAR' TYPES, HEIGHTS
9998  REM           AND WEIGHTS
9999 :
10000  DATA WARRIOR,6,200
10010  DATA WIZARD,6,150
10020  DATA THIEF,5,140
10030  DATA DELVER,5,160
```

Program 3.3

Line-by-line analysis

Line 10: As I said in the line notes for the last program, many computers do not require very small arrays to be DIMensioned. If you decide to dimension *all* arrays in advance I would suggest that wherever possible you use each array more than once. It would be possible, for example, to make the arrays CT$(), H1() and H2() rather larger than they are here, and then re-use them in the body of the game as 'object arrays' (see Chapter 5).

Lines 100–130: For general details see the same lines in Program 3.2. By the way, if you want to translate these programs for a different computer, then for lines 10 and 120 in particular check your own manual to see if multiple DIM and READ commands can be used in the form given here, or whether they need to be DIMmed and READ individually.

Lines 200-290: This is almost a direct copy of the lines 100-180 of Program 3.2. See the appropriate line notes.

Lines 300-320: Deal with the INPUT in a slightly different way from that used in Program 3.2 in that only *one* digit is collected for the Character Type. There's no special reason for choosing one method over the other – I just like experimenting.

Lines 400-480: These statements set up the whole of the screen display shown in Fig. 3.3 *except* the last two lines.

```
BEOWULF YOU HAVE 48 RATING
POINTS TO BE DIVIDED BETWEEN 6 CHARACTER
QUALITIES (USING WHOLE NUMBERS ONLY!)

EACH QUALITY MUST BE GIVEN AT LEAST 1
POINT.  NO QUALITY MAY BE GIVEN MORE
THAN 12 POINTS!

(1) STRENGTH          (4) SKILL

(2) HEALTH            (5) WEALTH UNITS/10

(3) INTELLIGENCE      (6) LUCK

ENTER POINTS FOR 1 AND PRESS RETURN 6
```

Fig. 3.3. Printout of lines 400–500 from Program 3.3.

Line 490: Sets up a standard FOR...NEXT loop which will be executed 6 times.

Line 500: Produces the last but one line in Fig 3.3. Note that the value of X actually prints out as part of the query.

Line 510: Checks that the player has entered a number between 1 and 12, and that it is an integer (a whole number). If any number not considered 'legal' is entered the player is advised of the fact (see the last line Fig 3.3) and line 500 is repeated.
Note: When line 500 is repeated (after line 510 or 520) the *wrong* input is left intact at the end of the query line with the cursor sitting over the left-hand digit. I did think of erasing the incorrect input on the grounds of tidiness. But then it occurred to me that the player might not automatically realise *why* the input was wrong. With this in mind I have left the original input intact so that the player can consider it in the light of the instructions at the top of the screen.

Line 520: Checks whether the player is in danger of using up all his rating points too soon. The variable CH holds all the points entered to date. To this is added the current input (CT(X)) plus the number of qualities still to be rated (as 6−X) and the total is compared with the total points allowed, in this case 48. If the total is greater than 48 then the player is told how many rating points he has left and the program returns to line 500.

Lines 530-540: If the input is satisfactory in all respects then it is

added to CH - the total number of rating points used to date. Next, 32 blank spaces are printed on the error message line, to erase any earlier message, and the query line is also blanked out entirely - including the last input. Then, if necessary, the program returns to the start of the loop.

Lines 600-620: At this point the CT() array holds only six values - those included in the screen display. For the sake of compatibility with other programs in this chapter I have now moved the values in CT(5) and CT(6) - Wealth and Luck - into CT(7) and CT(8) respectively, and transferred the Height and Weight values in H1(CN) and H2(CN) into CT(5) and CT(6). The Character Type name is again transferred from CT$(CN) to CT$.

Lines 10000-10030: These DATA statements contain the name, height and weight for each of the four character types, to be stored (temporarily) in the arrays CT$(), H1() and H2().
Note: Because all the DATA in lines 10000-10030 is collected at the start of this program I do not need the 'clear out' subroutine found in Program 3.2, lines 600-630.

By the way, if you wish to POKE the character ratings and name into memory as before then add these lines at 700-770:

```
700  FOR X = 1 TO 8
710  POKE 699 + X,CT(X)
720  NEXT
730  POKE 712, LEN (CT$)
740  FOR X = 1 TO  LEN (CT$)
750  POKE 712 + X, ASC ( MID$ (CT$,X,
     1))
760  NEXT
770  POKE 710,CN
```

Program 3.4: Computer-set character type and ratings

This last character generator, actually the first one I ever wrote, comes the closest to duplicating the start of a game of *Dungeons and Dragons*. The rather unusual random number sequence in line 170 is taken from the 18-sided dice often used by board-gamers, and its value may be changed to suit your own needs.

The basic intention of this program - which actually executes very quickly, despite its apparent complexity - is to create wholly fictitious characters in a totally random fashion. This it does very

well; in fact my wife and I had great fun (when it was first written) assigning fantasy characters to all our acquaintances. It can be quite amusing when your boss - in real life six feet tall with a definite paunch - turns up as a three-foot Hobbit with a low IQ!

But why revert to a computer-controlled character creator after singing the praises of a player-controlled program? In the first place this program allows an even wider range of characters than did Program 3.3, enhancing the challenge factor of the game. Secondly, the player is still allowed a measure of control over the character, for this routine would appear at the start of an adventure, and if the player doesn't like the character the computer has generated he can always restart the program and hope for a better one next time.

As for disadvantages - well, to be honest, I can't think of any.

```
]LIST

1  REM *****  COMPUTER-SET CHARS'  **
     ***
2 :
3 :
8  REM *** SET UP CT() ARRAY
9 :
10  DIM CT(10)
20  FOR X = 1 TO 9 STEP 2
30 XC =  INT ( RND (1) * 18) + 1:YC
     INT ( RND (1) * 18) + 1
40  IF XC < 3 OR YC < 3 THEN 30
50 CT(X) = XC:CT(X + 1) = YC
60  NEXT
70  FOR X = 1 TO 9: PRINT CT(X): NEXT

86 :
87 :
88  REM *** MODIFICATION ROUTINES
89 :
97 :
98  REM *** RE-SET KIN TYPE
99 :
100 CT(9) =  INT (CT(9) / 3) + 1
110  IF CT(9) > 1 THEN CT(9) = CT(9)
     1
197 :
198  REM *** RE-SET HEIGHT & WEIGHT
199 :
```

```
200  IF CT(5) < 4 THEN CT(5) = 4:CT(6
     ) = 110: GOTO 300
210  IF CT(5) < 7 THEN CT(5) = 5:CT(6
     ) = 130: GOTO 300
220  IF CT(5) < 10 THEN CT(5) = 5:CT(
     6) = 160: GOTO 300
230  IF CT(5) < 13 THEN CT(5) = 6:CT(
     6) = 190: GOTO 300
240  IF CT(5) < 16 THEN CT(5) = 6:CT(
     6) = 220: GOTO 300
250 CT(5) = 7:CT(6) = 250
296 :
297  REM *** MODIFY RATINGS ACCORDING

298  REM          TO KIN TYPE
299 :
300 A = CT(1):B = CT(2):C = CT(3):D =
     CT(4):E = CT(5):F = CT(6):H = CT
     (8):I = CT(9)
310  IF I = 2 THEN A = A * 2:B = B *
     2:E = E * .7:F = F * .8
320  IF I = 3 THEN B = B * .7:C = C *
     1.5:D = D * 1.5:E = E * 1.25
330  IF I = 4 THEN A = A / 2:B = B *
     2:D = D * 1.5:E = E / 2:F = F /
     2
340  IF I > 5 THEN A = A / 2:C = C *
     1.5:D = D * 1.5:E = E / 3::F = F
      / 4:H = H * 1.5
360 CT(1) = A:CT(2) = B:CT(3) = C:CT(
     4) = D:CT(5) = E:CT(6) = F:CT(8)
      = H
396 :
397  REM *** RE-SET CT(1) - CT(8)
398  REM          AS INTEGERS
399 :
400  FOR X = 1 TO 8
410  IF CT(X) -  INT (CT(X)) < .5 THEN
     CT(X) =  INT (CT(X)): GOTO 430
420  IF CT(X) <  > INT (CT(X)) THEN
     CT(X) =  INT (CT(X)) + 1
430  NEXT
497 :
498  REM *** GET NAME FOR CHAR'
499 :
```

```
500  PRINT "[HOME][DOWN * 6]PLEASE EN
     TER A NAME FOR YOUR CHARACTER"
510  INPUT "AND PRESS <<RETURN>> ";NA
     $
597 :
598  REM *** POKE ARRAY INTO 'SAFE' R
     AM
599 :
600  FOR X = 1 TO 9
610  POKE 700 + X,CT(X)
620  NEXT
647 :
648  REM *** PLUS LENGTH OF NAME
649 :
650  POKE 712, LEN (NA$)
697 :
698  REM *** AND NAME (AS ASCII CODES
     )
699 :
700  FOR X = 1 TO  LEN (NA$)
710  POKE 712 + X, ASC ( MID$ (NA$,X,
     1))
720  NEXT
730  FOR X = 1 TO 10: PRINT  PEEK (70
     0 + X): NEXT
740  FOR X = 1 TO  PEEK (712): PRINT
      CHR$ ( PEEK (712 + X));: NEXT
```

Program 3.4.

Line-by-line analysis

Line 10: As before, this line may be omitted if you wish.

Lines 20-60: This FOR...NEXT loop fills the array CT() with 10 random numbers in the range 3 to 18. I chose not to allow any characteristic to have a rating less than 3, because this would give the player an unfair disadvantage.

Line 70: Is included for checking purposes only (to see that you're really getting a set of *random* values). In a proper game version of the routine this line should be removed.

Lines 100-110: CT(9) contains the value for the *kin type* - Human, Dwarf, etc. As I was only using five kin types (see next program) I have modified this value to fit within the range 1 to 6.

(The sixth value is allowed for in line 200 of the Status Display program which follows.)

Lines 200-250: In these lines the height and weight for each character is preset to fit with a reasonable set of height/weight ratios. Since the control value is in consecutive blocks it is quicker to jump to the next section once the two CT() values have been set rather than go on 'falling through' the rest of the IF statements.

Lines 300-340: Here all relevant character qualities are modified to fit particular kin types. If CT(9)=1 then the kin type is Human and the character keeps his given ratings. Leprechauns, on the other hand, being traditionally known as 'the little people', have their height reduced by two-thirds, their weight reduced by three-quarters and their strength reduced by half. This is compensated for, to a degree, by their skill, intelligence and luck being increased by a half (see line 340). When I first wrote this program – using an APPLE II+ (which allows up to 239 characters per program line) – all the CT() values were altered directly in this section of the program. Lines 300 and 360 are made necessary by the 80 character line limit on the C64.

Lines 400-430: Another FOR...NEXT loop, which turns all CT() values except CT(9) – which is already an integer – into whole numbers. These lines are only needed if you intend POKEing the values into RAM.

Note: In many machines – including the C64 – turning a floating point (i.e. decimal) number into an integer means that it will always be rounded *down*, so the integer value of both 3.01 and 3.99 would be 3. This is avoided in lines 410 and 420, where values below X.5 – where X is the main value – are rounded *down*, and values of X.5 and above are rounded *up*.

Line 500-510: The familiar routine for getting the player's (fantasy) name as NA$.

Lines 600-620: Almost the last FOR...NEXT loop, to store all the values of CT() up to CT(9) in RAM. (CT(10) is not in practical use at the moment – see next program.) If you prefer to keep using the CT() array ignore these lines.

Line 650: POKEs the length of NA$ to 712 for use by the Character Display program coming up next.

Lines 700-720: Definitely the last loop of the routine proper – again to POKE NA$ into memory using the ASCII values.

Lines 730-740: A final check routine, for demonstration purposes only. Line 730 prints out the character ratings followed by (in line 740) the character name – complete with reversed letters!

Program 3.5: Character status display and printout

The last program in this chapter is the Status Display I mentioned earlier. As given here it forms a separate program from the actual Character Generators (though it does a tiny bit of creating itself in lines 100 to 160). The program was originally written to display the results of Program 3.4 at the end of each section of a modular game. The game was in a number of separate parts, and when each part was successfully completed – or when the hero met his doom – the game module would POKE the rating values into RAM, the Status Display program would be loaded (in place of the game module) and the Status Report would appear.

That was a while back, before I discovered most of the space-saving tricks you'll find in later chapter, and the version of the program which appears here is intended as part of the master

```
BEOWULF, OF THE HOBBITS
THESE ARE YOUR CHARACTERISTICS:-

SEX: MALE                TYPE: APPRENTICE

STRENGTH: 6              HEALTH: 10

INTELLIGENCE: 5          SKILL: 7

HEIGHT: 5 FEET           WEIGHT: 140 POUNDS

WEALTH: 8 GOLD COINS

MAX. WEIGHT YOU CAN CARRY: 60 LBS

YOUR LUCK RATING IS: 10

AS AN APPRENTICE YOU MAY CHOOSE TO STUDY
SWORDSMANSHIP OR SORCERY, BUT NOT BOTH.

PRESS <<RETURN>> TO CONTINUE
```

Fig. 3.4. Printout of lines 400–600 from Program 3.5.

program which could be called up at any point in the game using the command STATUS, for example.

Before giving the listing I should just point out that this program uses the whole of the CT() array including CT(9), which only exists as a positive value in Program 3.4. To use the Status Display with other modules, slight amendments may need to be made to allow for characteristics which have not yet been set (this applies particularly to Program 3.1). You will also see that Program 3.5 assumes that all character ratings have been transferred to the area of RAM from 700 onwards.

The overwhelming advantage of including the Status Display in any adventure is, I hope, abundantly clear. It is one of the best ways I know of giving a player the necessary sense of involvement and progress lacking in so many games available at the moment. Just consider, for a moment, the comparative effect of the display in Fig. 3.4 as against a message like

YOU HAVE SCORED 165 POINTS

YOU HAVE COMPLETED 20% OF THE GAME

YOU NOW HAVE 350 GOLD COINS

```
]LIST

1  REM *****  CHAR' STATUS DISPLAY  *
     ****
2 :
3 :
8 : REM *** CHECK 710 FOR CHAR' TYPE
9 :
10  IF  PEEK (710)       0 THEN 100
17 :
18  REM *** IF NOT SET THEN SET IT
19 :
20 A =  PEEK (701):B    PEEK (703)
30  IF A > 10 AND B   10 THEN  POKE 7
     10,1: GOTO 100
40  IF A < 11 AND B   10 THEN  POKE 7
     10,2: GOTO 100
50  IF A < 8 AND B > 8 THEN  POKE 710
     ,3: GOTO 100
60  IF A > 12 AND B > 12 THEN  POKE 7
     10,4: GOTO 100
70  IF A > 7 AND B > 7 THEN  POKE 710
     ,5: GOTO 100
```

```
80  POKE 710,6
97 :
98  REM *** GET CHAR' TYPE FROM 709
99 :
100  ON  PEEK (710) GOTO 110,120,130,
     140,150,160
110 TY$ = "WARRIOR": GOTO 200
120 TY$ = "WIZARD": GOTO 200
130 TY$ = "THIEF": GOTO 200
140 TY$ = "WARWIZARD": GOTO 200
150 TY$ = "DELVER": GOTO 200
160 TY$ = "APPRENTICE"
197 :
198  REM *** GET KIN TYPE FROM 709
199 :
200  ON  PEEK (709) GOTO 210,220,230,
     240,250
210 KI$ = "OF THE HUMAN KIN": GOTO 30
     0
220 KI$ = "OF THE DWARVES": GOTO 300
230 KI$ = "OF THE ELVES": GOTO 300
240 KI$ = "OF THE HOBBITS": GOTO 300
250 KI$ = "LEPRECHAUN"
297 :
298  REM *** RETRIEVE CHAR' NAME
299 :
300 NA$ = "[RVS]"
310  FOR X = 1 TO  PEEK (712)
320 NA$ = NA$ +  CHR$ ( PEEK (712 + X
     ))
330  NEXT
340 NA$ = NA$ + "[OFF]"
397 :
398  REM *** DISPLAY CHAR' STATUS
399 :
400  PRINT "[CLEAR][DOWN * 3]"NA$;
410  PRINT ", "KI$: PRINT "[DOWN]THES
     E ARE YOUR CURRENT RATINGS:-"
420  PRINT "[DOWN]SEX: MALE"; TAB( 22
     );"TYPE: "TY$
430  PRINT "[DOWN]STRENGTH: "; PEEK (
     701); TAB( 22);"HEALTH: " PEEK (
     702)
440  PRINT "[DOWN]INTELLIGENCE: "; PEEK
     (703); TAB( 22);"SKILL: "; PEEK
     (704)
```

```
450  PRINT "[DOWN]HEIGHT: "; PEEK (70
     5);" FEET"; TAB( 22);"WEIGHT: ";
      PEEK (706);" LBS"
460  PRINT "[DOWN]WEALTH: "; PEEK (70
     7);" GOLD COINS"
470  PRINT "[DOWN]MAX. WEIGHT YOU CAN
      CARRY: "; PEEK (701) * 3;" LBS"

480  PRINT "[DOWN]YOUR LUCK RATING IS
     : "; PEEK (708)
490  IF  PEEK (710) = 6 THEN  PRINT "
     [DOWN]AS AN APPRENTICE YOU MAY C
     HOOSE TO STUDY"
500  IF  PEEK (710) = 6 THEN  PRINT "
     SWORDSMANSHIP [RVS]OR[OFF] SORCE
     RY, BUT NOT BOTH."
597  :
598  REM *** WAIT FOR PLAYER
599  :
600  PRINT "[DOWN * 2]PRESS <<RETURN>
     > TO CONTINUE ";
610  GET Z$: IF Z$ = "" THEN 610
620  END
```

Program 3.5.

Line-by-line analysis

Line 10: You will remember that in the last program we went as far as defining the Character Type along with all the other details. This line checks whether this characteristic has, in fact, been stored. If it hasn't then the players may find that their character is slightly modified on the basis of their current *strength* and *intelligence* ratings because this program allows for six character types rather than the previous four. **Note:** For truly progressive characters omit this line so that the Character Type can be re-evaluated (see lines 20-80) as the player gains or loses strength and intelligence points.

Lines 20-80: These lines set the Character Type and POKE the relevant code into 710 for future reference. You will see that the whole subroutine depends upon the values stored in 701 (strength) and 703 (intelligence). These values are transferred to variables A and B for speed of execution and to save a little space. Even allowing that keywords like PEEK are 'tokenised' so that they only take up one byte in memory, IF A > 10 still has to be shorter, in both respects, than IF PEEK(701) > 10. **Note:** The values used to decide

each character type have been set fairly arbitrarily in this example and may be altered as you see fit. At the moment you only qualify as an Apprentice if you don't fit into any other category.

Lines 100-160: Rather than store the Character Type itself in memory I've included the six strings in the display routine. This also makes it easier to change the Character Type as a person progresses. These lines merely set up TY$ according to the value in 710.

Lines 200-250: This section follows the same method as seen in lines 100-160, only this time we are setting the character's Kin Type.

Lines 300-340: This routine collects the player's name from memory for the display. Although they won't actually make any difference if left in, lines 300 and 340 may be deleted if you have stored the REVERSE ON/OFF bytes with the name as shown in previous programs.

Lines 400-480: These lines print out most of the chart shown in Fig. 3.4. The last section, which will only apply if the player falls into one specific category *and* if you want to include some kind of option, is produced by lines 490-500.

Lines 600-620: A simple method of legally 'hanging' the program until the player has had time to digest the contents of the Status Display (thus providing a means of 'freezing' a game which includes any *real* time action, incidentally). You will notice that while the player is asked to press the RETURN key, pressing any key will in fact have the same effect. Some people may find this a bit off-putting, so if you want to be more precise either change the wording in 600 to PRESS ANY KEY TO CONTINUE or alter the second statement in line 610 to IF Z$ <> CHR$ (13) THEN 610.

Program 3.6: George and the dragon

The last routine I want to include in this chapter shows how to arrange a very simple form of combat between the player's character and a second or third level opponent – in this case George and a dragon. It also serves to show how the character ratings, once set, can be altered from within a program. If you are using this routine in a program with a character generator then for all the player's ratings you should use the values you already have in an array, or POKEd into memory.

By the way, you might like to notice how the player's LUCK rating is being used here. The player's *combat* strength is reached by adding his actual strength rating to his luck rating to give extra weight to his fighting ability. Even so, at first sight it might seem that the dragon has an unfair advantage over George in that its skill rating (ML) is only 3 points lower than George's (SL) whilst it's strength rating (MH) is 11 points higher than George's combat strength (PT). In actual fact I originally gave the dragon only 20 strength points, but after running the program two hundred times I found that the dragon was only winning five per cent of the confrontations! I therefore raised the dragon's strength to the current figure to give it a fairer chance!

This program will run by itself. And lines 35 and 55 have been included so you can watch how the battle is going. In a proper game program, however, you may choose to omit these two lines and simply display the result.

```
10 SL = 5: SH   12: LK   3: PT   SH + LK: ML
   = 2: MH = 26
20 PS  = INT ((SL * ((RND(1) * 6) + 1) + PT)
   / 6)
30 MH = MH   PS: IF MH < 1 THEN 100
35 PRINT "DRAGON   "MH
40 MS  = INT ((ML * ((RND(1) * 6) + 1) + MH)
   / 6)
50 SH = SH   MS: IF SH < 1 THEN 150
55 PRINT "GEORGE = "SH
60 GOTO 20
100 PRINT:  PRINT "WELL DONE GEORGE - YOU'VE
    KILLED THE(4 spaces)DRAGON!": END
150 PRINT:  PRINT "WHOOPS! DRAGONS 1, YOU'VE
    LOST.  R.I.P.": END
```

Program 3.6.

Line-by-line analysis

Line 10: The meanings of the variables used in this line are as follows: SL stands for player's SkilL, SH for player's StrengtH and LK for player's LucK. PT, as I explained before, is the Player's combaT strength. The dragon, being a second level character (because he can actually *kill* the player's character), does not get a combat rating and has to make do with a limited skill rating (ML) and a fairly hefty strength rating (MH).

Line 20: The next variable, PS, stands for Player'S hit and is calculated by the same means used for the Monster'S hit in line 40. Thus the effect of each character's blows are found by multiplying their skill rating by a random number between 1 and 6 (as if we were throwing an ordinary dice). We then add on the character's strength (combat strength for the player, ordinary strength for their opponent), divide the total by 6, to make the combat last a bit longer, and finally round the result down to the nearest whole number. Please note that there are *ten* brackets in this line and in line 40. They must *all* be entered in the correct place to make the calculation work correctly.

Lines 30-35: Having allowed the player to go first we now deduct the effect of his blow (the value of PS) from the dragon's strength rating and check whether that rating has fallen to 0 or below. If it has, usually after about three or four rounds, then the program goes to line 100 to display George's victory message. If it hasn't then the dragon's *modified* strength rating is displayed and the dragon gets to take a chunk out of George.

The fact that George always goes first, so that the dragon never gets to attack him with its full strength is, of course, greatly to George's advantage and the reason why the dragon needs such a high strength rating to have any real chance of winning.

Lines 40-55: Given that it is now the dragon's turn to attack George, these lines are a direct copy of lines 20-35 with the variables altered accordingly.

Line 60: If George is still alive after line 40 – if his *strength* rating, *not* his combat strength rating, is still higher than zero – then the program returns to line 20 for another round.

Lines 100 and 150 print out the victory messages for George and the dragon respectively. It should be noted, here, that unless George has been particularly lucky then his strength rating will be very low at

this point – over four hundred combat sessions his average strength at line 100 was 2! This should be allowed for in a proper game setting by giving him some means of getting his strength back before he is required to undergo any further strenuous physical activity – an enforced rest (in game time, not real time) or a healing potion of some kind according to the nature of the storyline.

Again, if you are using a character generator and your player survives the combat, his new strength rating (SH) should be used to alter the original list of character ratings before the game continues.

Where to go from here

Well, that brings us to the end of this first 'programming chapter'. There are plenty more programs for you to test and use in your own games if you so wish. Most of all, though, do try *experimenting* with these programs. Some of them are based on the latest techniques in programming (at the time of writing) but that in no way implies that someone, somewhere, isn't already thinking up better, faster, more efficient methods of doing the same tasks. That someone could be you!

You will already have seen that some routines in this book do not give a final result unless extra lines are added. This is because they have been written as *modules*, to be linked together to get the exact program you want to produce. So don't be afraid to link up different routines to see what kind of result you get. It may not always be exactly what you expected, but if this is the case try to work out *why* you didn't get what you were after.

As you study the various programs you'll find almost every kind of BASIC routine you could need for *any* kind of program – string checks, PEEKing and POKEing, etc. If there's anything you don't understand straight away, enter the program and RUN it to see what it does, and then check it against the line-by-line analysis. If it has POKEd something into memory, try using something like the checker routine in lines 730-740 of Program 3.4 to see *what* it has POKEd. (You'll find a lot more about 'reading' the memory in Chapter 6.)

The best way I know of pulling a program to pieces is to alter it in small ways here and there. Try leaving out a line or two and see what difference it makes. If you can't follow what the variables are doing, then add a line that will print out what is happening to them. And if you think you could write a better piece of coding then by all means do so!

Chapter Four

O.K. Bugsy - We Know You're in There!

You're in one of the bedrooms of a deserted house. As you crouch by the window a rat runs over your foot. Although it's night-time, and the single light-fitting is broken, the room is as bright as day in the glare of the searchlights outside. You wait. And after a moment or two a voice, crackling through a well-used megaphone, shouts 'O.K. Bugsy, we know you're in there. Come out now, or we'll blow you out!'

Where are you? How did you get here? And why are they calling you Bugsy?

The answer to all these questions can be found in *Superspy*, a yet-to-be-completed tale of espionage folk. You're in the building which stands over the subterranean headquarters of the mad Professor Geri (though you probably don't yet know what's in the basement). You got there by following the advice of someone you should never have trusted in the first place. And the police outside are calling you Bugsy because they, too, are following a false trail.

Now let me answer those first two questions another way!

Where are you? In 'room' 56.

How did you get here? By moving from room 47, through rooms 51, 52 and 53 to your present location. In other words you followed the trail through the fiendishly cunning map devised by the writer of *Superspy*. Which brings us very neatly to the subject of this chapter – mapping out an adventure game.

Picking your spot

If you've already developed a storyline for your adventure, then choosing the landscape within which the action will take place should come fairly easily. Some writers, on the other hand, may find it easier to start with a setting for their adventure and develop the

storyline from there, which is fine. Indeed, if you haven't already used this method why not give it a try? It may be the one which works best for you.

But how do you choose the *best* location for an adventure? That may sound like a silly question but it does have a point. The best place to set a story is 'in the right place', but this isn't always the most obvious place.

First, it's as well to start thinking about a location for your story in the same way that you approach the story itself – with a completely open mind.

It might seem, at first, that your whole story can be set in a single general location – a country house, a space ship, or wherever – as are *Cranston Manor, Star Cross* and several other successful games. In *Deadline* the entire action is confined to the vicinity of a fairly luxurious, though otherwise perfectly ordinary house, with not a trapdoor or secret cupboard in sight. But on the other hand don't be afraid to spread yourself around a bit if the mood takes you. Remember that many popular books and films owe a good deal of their effect to the constantly changing locations of the action. Of course, such changes won't by themselves make for a good story. But if you think that your adventure can gain by moving all over England, or all over the world for that matter, then by all means try it.

'Hang on', says a voice in the background, 'how do I move the hero around like that without having a map as big as the living room floor?'

There are at least three methods which spring to mind immediately, and probably several more that I haven't yet thought of:

(1) Provide players with user-controlled transport such as a car or a motorbike, so that instead of using the normal GO SOUTH command they can use DRIVE SOUTH. In this way they will find it believable that they have travelled ten or twenty miles in one move (and from one 'room' to the next on your map), because the act of getting into a car and driving implies a sense of distance.

(2) The act of travelling can become even more interesting when players don't know where they are travelling to, especially if you want them to cover a substantial distance, say fifty miles or more. In this case the player may be brought to a railway station or an airport by normal means (this could be the start of the adventure, perhaps), and then offered a ticket for the train or plane which is about to

depart. Using a little routine that you will find at the end of Chapter 5 you can ensure that the player is forced to make a choice before he's had a proper chance to consider the possible consequences. If you don't want to include the element of surprise the same event can be re-written so that the player has to get hold of a rail or air ticket before he can progress to the next stage of the game (not so easy if he is short of cash or his pocket has just been picked!).

(3) In the third alternative the player might be tricked into travelling before he has a chance to avoid it. Thus he might be encouraged to walk through a door which leads straight into the back of a truck, or into an interstellar version of the Space Shuttle. In a flash the door of the vehicle is closed behind him and locked, and the text display shows that he has been transported from one side of the country, or one side of the galaxy, to the other!

It comes down to the fact that in an adventure game the entire universe is your oyster.

All the heavens in a grain of sand

Having offered you the universe, we now have to come down to earth for a moment and consider another kind of space - RAM space - and its limitations.

One of the main reasons for writing this book was to encourage would-be adventure writers by showing how great adventures can be squeezed into surprisingly small (memory) spaces. This will become much clearer in Chapters 6 and 7. But for the moment it is necessary to point out that even when RAM space is used to the best possible effect, it is still measured in very finite amounts. So the next step is to decide roughly how big your adventure can be.

The first thing we need to know in reckoning the overall size of a program is whether it will be stored on tape or on disk. Information is written to and collected from different storage media in different ways. Cassette tapes, for instance, can only hold 'sequential' files. That means the information is written and read as one continuous stream, so you cannot jump straight to a particular piece of information on the tape with any degree of accuracy. Instead you must read through a file from the beginning until you find the information you want. Also the process of reading from, or writing to, a tape file is extremely slow.

Disk storage, on the other hand, is extremely fast. Because the disk

is flat the read/write head can read the disk directory track (a kind of index to everything on the disk) and then move directly to the start of a particular file. (To understand what I mean think of the difference between trying to find the start of a chosen song on tape, and doing the same thing on an LP record.)

And this isn't the only advantage disks have over tape storage. Again, because of the way the information is stored, the Commodore disk drives allow for a total of *four* different methods of storage – sequential, block access (moving 256 bytes at a time), random access and relative access (a variation on Random Access). In all the last three methods information is stored in such a well-organised fashion that the computer can call up a single piece of information – or even a single byte (!) – from within a file without starting at the beginning. Instead, the read/write head is simply moved to the start of the required file, record or field and reads it or writes to it as required.

The difference between sequential and random access storage is clearly very important when deciding how big an adventure will be. The largest part of an adventure, with the *possible* exception of the master program, is the room description file. If this is stored on tape it must be fed into the computer before it can be accessed in the manner required in an adventure. If it is on disk, however, each room description can be called up directly from the disk when needed. Thus the amount of RAM space required for the room description file as a whole is no bigger than the size of just one or two records (depending on the size of your room descriptions). One of the longest adventures I have yet discovered – *Fantasyland 2041 A.D.* – actually uses *six* disk sides, one for the master program and five for room descriptions and graphics displays!

So, game size depends on three main factors:

(1) How much 'user RAM' you have in your machine
(2) Whether you are using cassette storage or disk drives
(3) Whether the game consists of a single unit or a number of linked modules.

This last factor concerns the method of laying out your program. If it is all in one piece – if the whole program is loaded at the start of the game – then factors (1) and (2) are particularly important. But if the game is divided into separate sections, each with its own master program and room descriptions, etc., then you can overcome many of the limitations of limited RAM space and cassette storage.

I wouldn't advise newcomers to try writing modular games

straight away, but once you have mastered the other aspects of the art it is well worth taking advantage of the freedom of action this approach offers.

The mushroom factor

So just how much room do you have in your computer? The answer, I'm afraid, is probably not as much as you thought.

The most common sizes for home computers on the market today are multiples of 8 (because they are 8-bit machines). Thus we have the Vic 20 (with minimum expansion) at the bottom of the table with 8K, followed by the smaller Spectrums and the Atari 600XL which are 16K machines. The BBC B, the original Dragon and the Electron are 32K machines. The standard Oric, the Atmos, the larger Spectrum and the unmodified Apple II series are all listed at 48K. And the Commodore 64 weighs in at 64K! These figures will already be familiar to most micro owners, I know. The reason I have listed them here is because they are, to say the least, highly misleading.

In practice (assuming that we are talking about text adventures only) the machines listed above have the following amounts of user RAM space:

Vic 20 (expanded to 8K) – about 6.5K
Spectrum 16K – about 9K
The Atari 600XL – 13K
The Electron – 20K
The Dragon 32 and the BBC B – 27–28K
The Oric 1 (without high resolution graphics) – about 46K
The Spectrum 48K – about 40K
The Apple II+ (minus high resolution graphics *and* DOS) – 50K
The Commodore 64 – a few bytes less than 38K

In point of fact the C64 has more than 64K, and all 48K micros are actually 64K machines! In other words they contain 64K bytes of memory space if ROM, computer-controlled RAM etc. is included. This is why, in the case of the Apple II+, you can find 50K of user space in a 48K computer!

So once we know roughly how much space our game has to fit into, we can begin to get some idea of how large it will be.

If we take the control program first, then it is fair to say that quite a sophisticated program can be made to fit into about 16K–20K

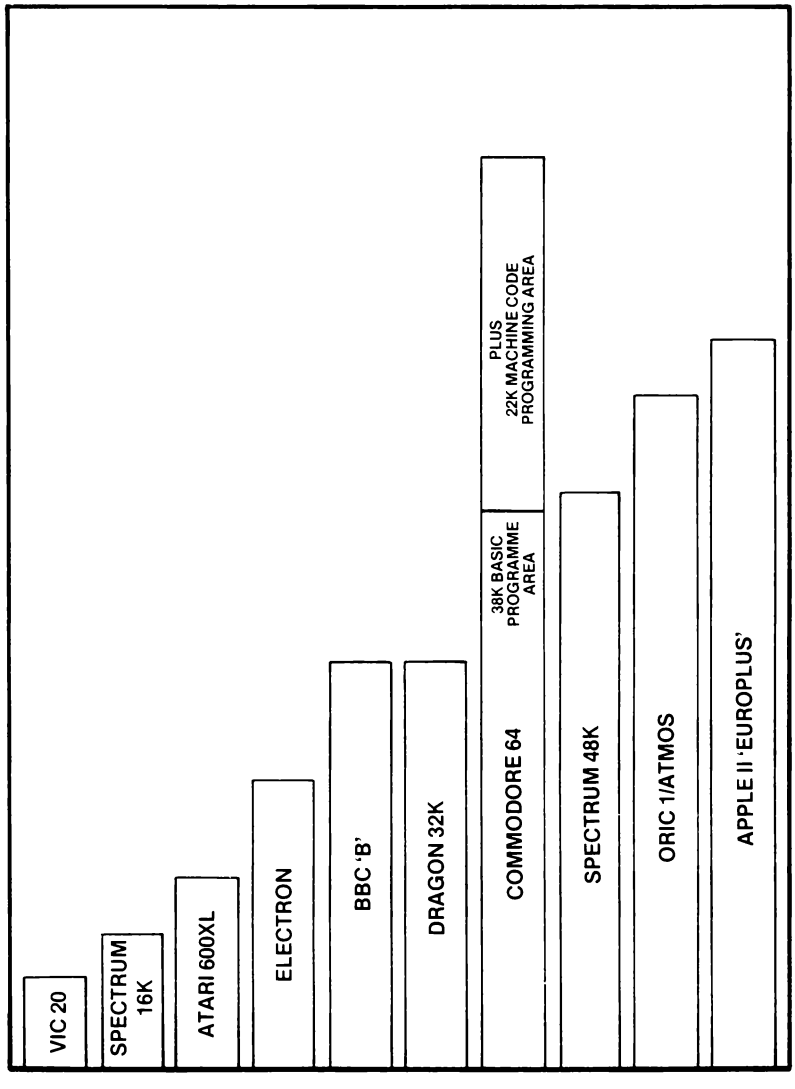

Fig. 4.1. Chart showing comparative size of user RAM in ten best-selling micros.

(using machine code). Unfortunately this is only the first step in calculating the total space needed. We must also allow storage for strings, numerical variables and arrays (though the latter can, in some cases, be made to fit into a much smaller area than usual – see Chapter 6). Then the computer itself will need some free RAM space in which to execute the program. And finally you will have to decide whether you want to include any graphics displays (see Chapter 10 for further discussion of this last point).

All in all the size of your adventure – assuming that *everything* is loaded into the computer at the start of the game – should be estimated on the basis of not more than 100 locations for every 16K of user RAM. In terms of the C64 that comes to around 200–230 rooms. As you become more proficient in programming techniques you may well be able to improve on these figures. But it can be very frustrating to work out all the details of a game and then find that a major portion of the coding has to be rewritten because it won't fit into the available space.

In short, unless you have the time and patience to constantly trim and polish your game, start out by *underestimating* the amount of space available; then, if possible, add extra touches to make the optimum use of your machine.

Which brings us to the next stage of preparing a game: the drawing up of an adventure map.

Where am I? Where am I going?

(**Note:** The word 'room' when used in relation to map-making is taken to mean one location within the adventure. It may be an actual room, or the cabin of an aircraft, a carriage on a train, a passageway, etc., etc.)

Map-making can be one of the most interesting parts of preparing an adventure game. Indeed, writers have been known to get so involved in the map-making that they nearly forget what the map is for. But the preparation of a map isn't only a way of having fun. It is an essential part of a good game.

You may have a particularly visual imagination and be able, with very little effort, to envisage a complete setting for your adventure before you ever get anything down on paper. If this is true for you then beware. Even now there are games on the market which fail to run properly because they don't follow a consistent pattern of movements. In fact I can think of one adventure where part of the game layout (as described on the packaging) is actually missing altogether! If the programmers had been working directly from a clearly laid-out map this *should* never have happened.

So what does an adventure map look like? Obviously it won't resemble anything you've ever seen in an atlas. In reality a completed map may appear rather boring – not much more than a set of boxes, containing brief notes and linked together by lines or connecting

boundaries. Yet these 'simple' layouts represent a good deal of genuine creative effort.

Broadly speaking there are just three sorts of adventure map:

(1) Boxes and lines
(2) Linked boxes
(3) Linked octagons

Let's look at each method in turn and consider their relative advantages and disadvantages.

The error trap

The boxes and lines method is probably the easiest way of preparing a map, and can be useful as a means of preparing the 'first draft' for a game. As the title of this section suggests, however, I have many personal reservations about using this method, and cannot recommend it with any great enthusiasm.

The most positive thing to say about box and line maps is that they allow the chart-maker a good deal of freedom. Each location on the map is noted down in a separate box, and the box is linked to its neighbours with a solid line (to indicate movement possible in either direction), a solid line with an arrow (one way movement only), a broken line (conditional movement), and so on as shown in Fig. 4.2. Landscape features around each location, if relevant, can be noted in the spaces between the boxes. Unfortunately the freedom of this kind of map is also its biggest potential hazard.

In the first place a map of this type can easily get out of hand. This is no great problem if you are still in the early stages of planning. But if your final map is spread out all over the place – with some rooms off in the middle of nowhere with only numbers on their sides to show neighbouring locations – things can get quite confusing.

The second objection is that box and line maps very often violate what I call the 'consistency rule'. This can be seen quite clearly in Fig. 4.2 in boxes 5, 6 and 7. In this area the player can move into room 6, a corridor, from either room 5 or room 7 by using the command GO NORTH. But what happens if he changes his mind and wants to go back? The command GO SOUTH, when used in room 6, has *two* possible destinations. If I actually want to progress (by moving to room 7) then all is well, if the program only allows movement from 5 to 6 to 7. But supposing I want to go back to room 4 to get the gun I previously ignored? The computer has no idea where I *want* to go,

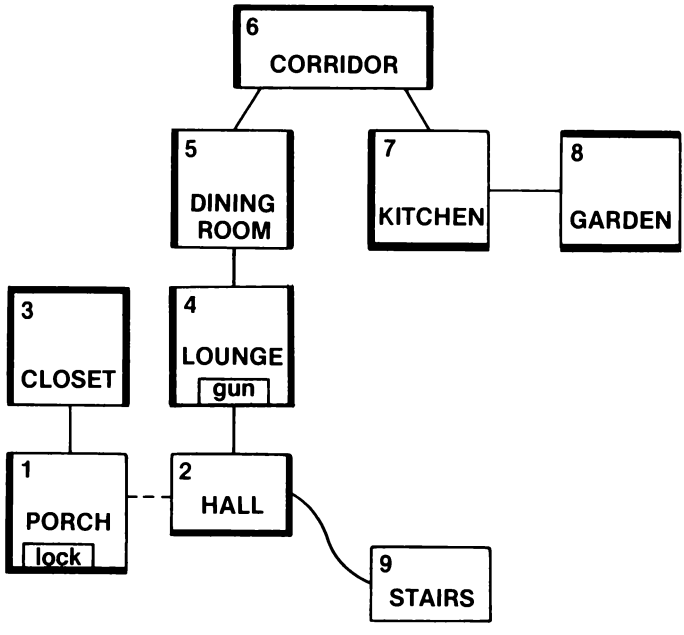

Fig. 4.2. Boxes and lines map.

only where the programmer has allowed me to go. Of course we could add an extra section to the command routine so that I am asked which way I want to go: LEFT DOOR OR RIGHT DOOR? But unless this choice will appear several times, the programmer has wasted space dealing with a situation that need never have arisen in the first place.

This kind of situation is also very confusing for the player, of course. If I'm in room A to start with, go north into room B and then go south and find myself in room C it is not immediately clear whether the program is still functioning correctly. Newcomers to adventuring, finding themselves in such a situation, might well assume there is a bug in the program and try to get a replacement.

As I said before, this method has its uses. But they are strictly limited.

Programs 4.1 and 4.2: Linked squares

The second method I want to discuss is far superior to the box and line approach, and may be seen as a limited version of the linked octagons system. Its main drawback, in fact its *only* real drawback,

is that it can lead the programmer into a common, and rather disastrous, error, which I will explain in a moment.

In order to use the linked squares method of map-making all you need, to start with, is a pencil and several sheets of paper marked out in fairly large squares. I say large squares because each square represents a single room and must, therefore, contain a room number, the room title, and the names of any objects or characters to be found in the room. Since each room has four walls you may have up to six possible exits and entrances – NORTH, SOUTH, EAST and WEST, plus UP and DOWN. Later on in this chapter you'll find details of how to mark these routes in a way that makes the map easy to read when it comes to programming the adventure.

And now for the problem I mentioned earlier. This usually arises only when the adventure writer uses the A × B grid system to be found in several other books on this subject. In the A × B grid below (Fig. 4.3) you'll see a ten by ten grid which gives a total of one hundred rooms. Since all rows and columns are equal in length it is

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Fig. 4.3. Linked squares map frame. If an 'unmodified calculated' move routine were used in conjunction with this map then off-the-edge errors could occur in any of the thirty-six squares *outside* the thick inner boundary line.

possible to use a small program that calculates which room you will end up in if you move in any direction. In this instance, the room directly NORTH of your current location will be CR (Current Room number)−10. The room directly SOUTH will be CR+10. The room directly WEST will be CR−1, and the room directly EAST will be CR+1.

```
]LIST

10  REM *****  CALCULATED MOVES IN
     10 X 10 GRID*****
20  :
30  REM    THIS ROUTINE ASSUMES THAT A

40  REM    MOVEMENT COMMAND HAS BEEN
50  REM    INPUT AS CO$
60  REM    CR=CURRENT ROOM NUMBER
70  :
80  REM ** ONLY READ LAST PART OF CO$

90  :
100   IF  RIGHT$ (CO$,4) = "EAST" THEN
      CR = CR + 1
110   IF  RIGHT$ (CO$,4) = "WEST" THEN
      CR = CR - 1
120   IF  RIGHT$ (CO$,5) = "NORTH" THEN
      CR = CR - 10
130   IF  RIGHT$ (CO$,5) = "SOUTH" THEN
      CR = CR + 10
140   RETURN
```

Program 4.1.

Line-by-line analysis

Lines 100–130: This very simple version of the calculated moves routine is based on a two-word (verb-noun) INPUT held as CO$. In each case the program simply reads off the relevant number of letters from the right-hand end of CO$ and performs a calculated move modification to the current value of CR (the player's location) in the fashion described earlier in this chapter.

Program 4.2: The wall around the world

But hold on for a moment. What happens if you're in a room with a

number ending in 1 and you move WEST – you move to the end of the previous row! And this kind of error can occur in any room on the edge of the grid. This can be dealt with quite easily, as in this second version of the program (Program 4.2), but again this system uses up valuable space to no good purpose. Indeed, as will be seen in Chapter 6, the use of *calculated* movements is a waste of time. Thus the linked squares map in Fig. 4.4 is just as satisfactory as, and rather less restricted than, the symmetrical layout in Fig. 4.2.

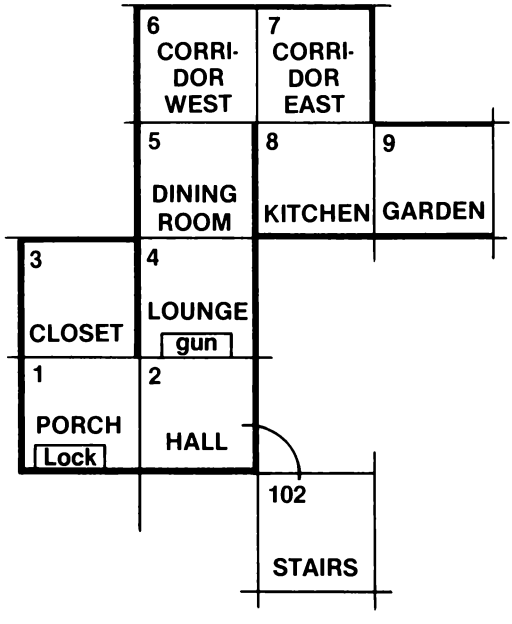

Fig. 4.4. Linked squares map.

```
LIST

10  REM *****  CALCULATED MOVES IN
     10 X 10 GRID (WITH MODIFICATION)
       *****
20  :
30  REM    THIS ROUTINE ASSUMES THAT A

40  REM    MOVEMENT COMMAND HAS BEEN
50  REM    INPUT AS CO$
60  REM    CR=CURRENT ROOM NUMBER
70  :
80  REM ** ONLY READ LAST PART OF CO$
```

```
90 :
98  REM ** CHECK FOR EDGE OF GRID
99 :
100  IF  RIGHT$ (CO$,4) = "EAST" AND
     CR / 10 =  INT (CR) THEN 300
110  IF  RIGHT$ (CO$,4) = "WEST" AND
     CR - (10 *  INT (CR / 10)) = 1 THEN
     300
120  IF  RIGHT$ (CO$,5) = "NORTH" AND
     CR < 11 THEN 300
130  IF  RIGHT$ (CO$,5) = "SOUTH" AND
     CR > 90 THEN 300
140  RETURN
197 :
198  REM ** IF MOVE IS LEGAL THEN DO
     IT
199 :
200  IF  RIGHT$ (CO$,4) = "EAST" THEN
     CR = CR + 1
210  IF  RIGHT$ (CO$,4) = "WEST" THEN
     CR = CR - 1
220  IF  RIGHT$ (CO$,5) = "NORTH" THEN
     CR = CR - 10
230  IF  RIGHT$ (CO$,5) = "SOUTH" THEN
     CR = CR + 10
240  RETURN
297 :
298  REM ** ADVISE 'ILLEGAL MOVE'
299 :
300  PRINT "YOU CANNOT MOVE IN THAT D
     IRECTION": PRINT
310  RETURN
```

Program 4.2.

Line-by-line analysis

Lines 100–130: Here the possibility of falling off the edge of the world is allowed for by calculating the player's location, *before* any move is made, in relation to the direction in which he wishes to go next.

In response to the command to GO EAST line 100 checks to see if the player is already on the eastern edge of the grid – in a room with a value that is a multiple of 10.

Line 110, in response to the command GO WEST, checks to see if

the player is in a room with a value that ends in 1 – that is, a room on the western edge of the grid.

Line 120, handling GO NORTH, checks for the northern boundary – rooms 1–10 inclusive.

Line 130, responding to GO SOUTH, checks whether the player is already on the southern edge of the grid – rooms 91–100.

If any illegal moves are entered execution passes to lines 300–310, the move is refused and the program RETURNs for a new command.

Otherwise execution moves on to lines 200–240 – a copy of lines 100–140 in the last program – the move is calculated, CR is altered, and again the program RETURNs for the next command. Obviously a real program would also deal with the possible consequences of each move, displaying a new room description etc., but these routines are intended for demonstration only.

Linked octagons

This last system is, I must confess, my personal favourite. Because each room has eight sides the number of entrances and exits is boosted to 10: NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, UP and DOWN. Using linked octagons does not increase the size of the room description file, but it will make your movement codes significantly larger. If you're working with a relatively limited amount of RAM space – less than 30K for instance – it might be as well to stick to linked squares when you plan your first adventure.

The main advantage of the linked octagons system, apart from the extended range of movements, is that it forces you to draw well-structured maps. If your map isn't properly laid out the results will become horrifyingly clear very quickly.

Some writers may regard the need to be so exact as a drawback. A more obvious disadvantage, however, is the fact that no one produces pads of paper marked out in octagons at the moment. It is a fairly simple matter to prepare these yourself, of course. Or you could prepare a master sheet and have it photocopied. This may sound rather extravagant, but unless you write your games very fast indeed you are unlikely to use up more than a dozen or so pages in a year.

... Twopence coloured

Having chosen the style of map you will use, the bare framework now has to be transformed into a 'living landscape'. This is easily done with a set of coloured felt-tipped pens. You will need a minimum of five, but it is worth buying one of the larger sets since they often provide as many as 30–40 pens for the same price as some of the smaller sets.

(**Note:** Although the next two maps are made up of linked octagons, exactly the same approach may be used for any other style of layout.)

The first thing to beware of when preparing a map is the urge to go too fast, to start writing things in – in ink – before you have roughed out all the details. Remember the creation of the map is one of the crucial stages in preparing an adventure. Go too fast, and you may find that when that sudden burst of creative ideas hits you, the map is already nearly complete. If that does happen you'll either have to re-draw the map (very frustrating), or leave out the gimmick that might have made the whole game rather special. Moreover, trying to turn two or more pages of badly-prepared map into a successful program will take far more time in the long run than you will need if the map is clear and well-organised. So let's get started.

There you are with a blank sheet of paper in front of you and a pencil in your hand. But where does the map start? In practice, since it's very unlikely that you'll manage to draw up a map to your complete satisfaction at the first attempt, it really doesn't matter too much *where* you put that first room. The centre of the page is probably as good as any. Now number and label that first location. And by the way, start numbering from 1, *not* from 0 – you'll see why when we start organising movement codes in Chapter 6.

From this point on the shape of the map is up to you. The direction which you plot for each move will depend entirely on what is happening in the adventure.

In Fig. 4.5 you'll see that I've already numbered twenty-two octagons, but only one has been labelled. Let's suppose that there is a large obstacle occupying rooms 5, 8, 9, and 12. I would start by blanking out this area and renumbering the octagons that will be used. (You see why everything is done in pencil to begin with!)

Next I develop routes away from room 1 which are consistent with the landscape in my adventure. If the action were set in a city, or in outer space, moves might follow the paths 1, 2, 4, 7, etc., or 1, 3, 6, 10. But if the adventure started in a forest, or an underground

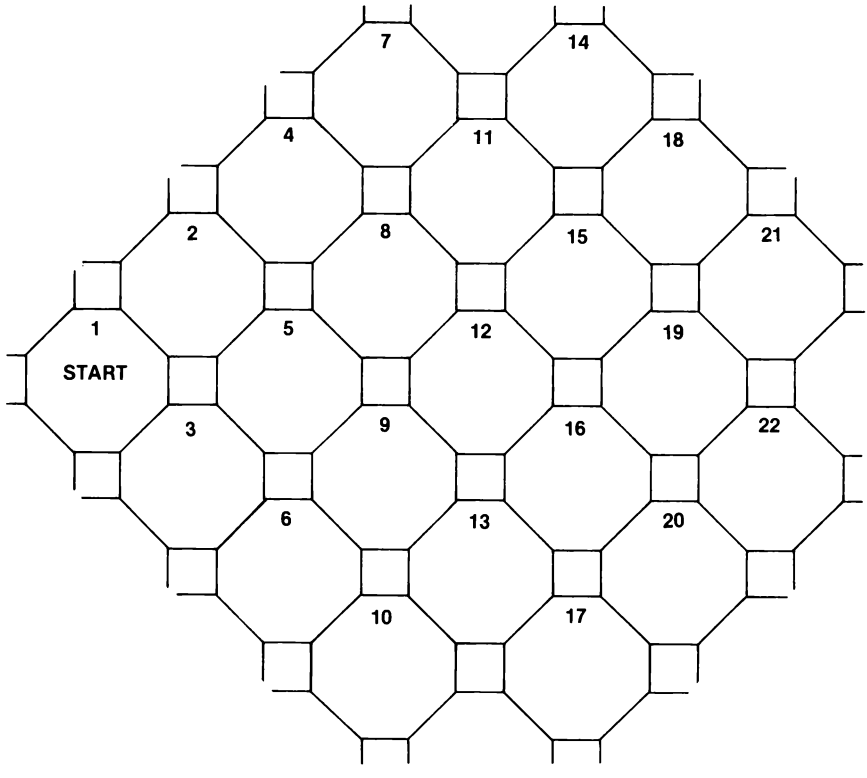

Fig. 4.5. Linked octagons map.

tunnel, I might start to alter the map again to give the twists and turns that would fit the situation.

Once I have the general layout of the map pencilled in to my satisfaction (with room titles) I can start placing objects and characters. It is now that the preparation of list 2 (in Chapter 2) will pay dividends. Before too long you should begin to see a landscape emerge on paper that really does bring your story to life.

But we're still working in pencil. Once you're satisfied with the overall layout of your map, you can begin to colour it in. This isn't a matter of making the map look any prettier, though if you're artistically inclined there's no reason why the map itself shouldn't be a work of art. Colouring your map, though not essential, is a way of making it more easy to follow when you start programming.

Colour coding

If you turn to Fig. 4.6 you'll see that it has altered quite significantly

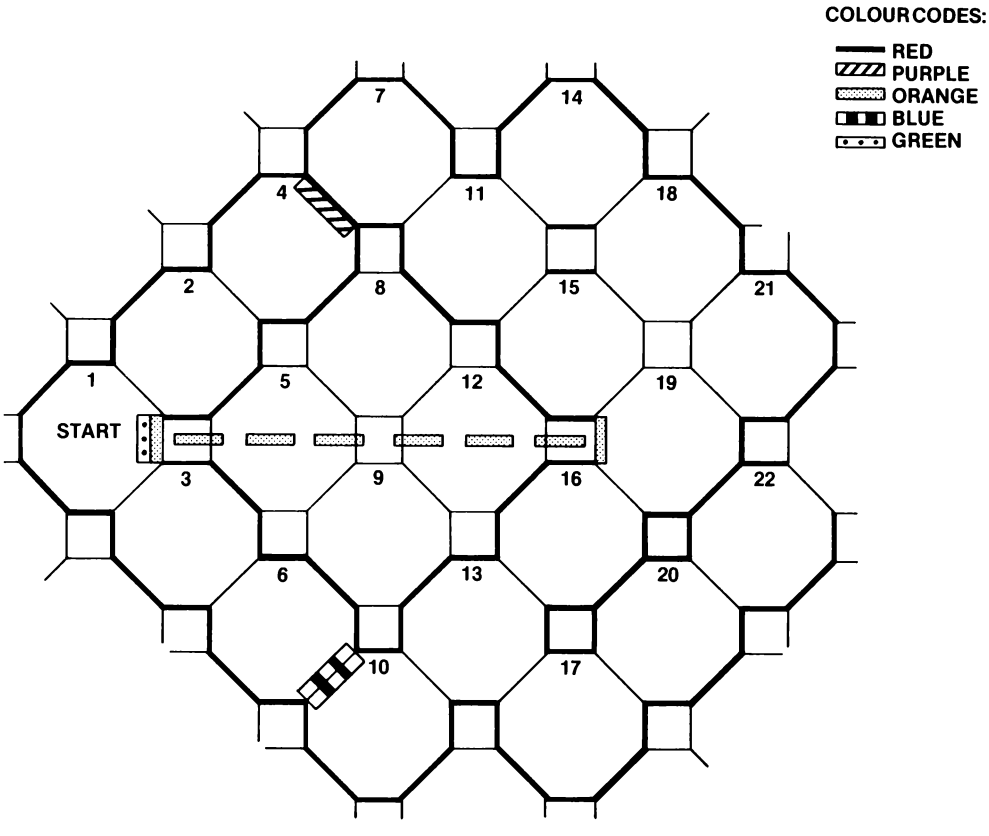

Fig. 4.6. 'Coloured' linked octagons.

from the simple framework in Fig. 4.5. Unfortunately it was not possible to provide a full-colour illustration of this map, and for that reason I have omitted room titles etc., for the sake of clarity.

In this example I've colour-coded seven different basic situations. Feel free to change the choice of colours and add more of your own if you wish. One word of warning though. Don't get carried away. The colour code system is designed to *simplify* your task. Use too many colours and you will defeat the purpose of the codes. And by the way, until you become familiar with the code that you are using it's a good idea to set out a chart of the colours and their meanings in one corner of *each* map.

Now for the actual colour codes. All room boundaries that *cannot* be crossed under any conditions should be marked in red on both sides. Thus the entire boundary of your map (except for any points where it may connect to another level) should be outlined in red, as well as any internal room boundaries that cannot be crossed. Note

that the coloured line should always be *inside* the walls of the room it applies to.

The reason for this last requirement can be seen in the case of rooms 11, 14, 15 and 18. The squares are not used as such, but they may provide links between rooms. In this instance it is possible to move from room 11 to room 18, but not from room 14 to room 15 (or vice versa).

So, red on both sides of a boundary means 'no way, no how'.

An item boxed in *red* is one which may *not* be picked up or used by the player, even though it might play an important part in the adventure. Such items do not need to be included in the 'object array' (see next chapter) as they will always be in the same place.

There may be occasions when you want to trap a player into going in a particular direction. In practice, then, the player can move *out* of a room but not back again – if they dive from a cliff edge into a river, say. In this situation the *purple* line marks the boundary of the room they may leave, while a *red* line marks the boundary of the room they enter and may not return from.

Other such situations might be doors with no handles or keyholes on the far side, lifts which break down when they reach a certain floor, airlocks which have been damaged so that they can only be used once, and so on.

Incidentally, it might seem that the purple line is unnecessary, and that the exit in a 'one way' situation could be left unmarked. The danger in taking this option is that the single red line may be misinterpreted when you come to program your adventure. Using the purple line as well is a form of insurance against such mistakes. A purple line, then, means 'one way only' and should always be coupled with a red line in the neighbouring room.

A purple line round an item in a room indicates that the item may be used only once. This will apply to many booby-traps, hand grenades etc.

Even if your game takes place on only one level there may be some situations where the player has to go *over* or *under* an obstacle. On my own map, since there is an obstacle in rooms 5, 8, 9 and 12, I've included a secret tunnel, marked in *orange*, between rooms 1 and 19 which may be entered from either end. (**Note:** If the tunnel had been one way only then the connecting line would have been marked in purple.)

The basic purpose of the orange boundary line, and connecter, is to indicate (a) a move to a different level in a multi-level game, (b) a connection between two rooms which requires the player to move

UP or DOWN rather than north, south etc., or (c) any link between non-adjacent rooms.

This last option is very necessary. The octagons are all the same size, but the locations they represent are not. Because of this, locations which *are* adjacent will not always appear to be so on your map, though the intervening spaces will normally be blank.

Items surrounded by an orange line come into a rather special category as they can move 'by themselves', as it were. This applies particularly to characters who move around the map independently of the adventurer – as in *The Hobbit*, for example. (Details of how to program such characters are given in the next chapter.)

Having used red and purple lines to indicate routes that are completely blocked off, in one direction at least, we now come to boundaries which allow 'conditional' movement. In such cases I use a blue line on one side of a boundary to indicate that the player may not move in that direction without satisfying some previous condition. If you look at Fig. 4.6 again you'll see that just such a situation exists at the border between rooms 6 and 10.

Going back to a previous example, this boundary line might represent the entrance to the Mad Professor's laboratory. If you remember, this door closes automatically once the player enters room 10. It can only be opened again if the player succeeds in paralysing Igor and finding the magnetic card. Thus the north-eastern boundary of room 10 is marked in *blue*. But that isn't all that happens in the laboratory, which means that we need a further code. Once Igor moves towards the player, the chemicals are knocked over and a fire starts. I have assumed that even if the player escapes from the laboratory, this fire will damage the door mechanism beyond repair. Thus room 10 may only be entered once – another conditional situation – and so *both* sides of the border must be marked in blue. (If you have blue on one side of a boundary only the border line itself may be indicated by a double-thickness line to avoid confusion. This is because crossing the line in one direction – from the non-marked side – does not require any action in the program. This point will become clearer when we come to movement codes in Chapter 6.)

Objects marked in blue also fall into the 'conditional' category. Such objects can only be accessed by the player if he already holds another object, has completed a certain task, or whatever.

The last situation I want to deal with by colour coding concerns hidden exits and hidden objects. In this case I use *green*. In a sense, the finding of hidden items is another conditional situation and

might be dealt with by using blue lines as described above. Personally I prefer to indicate the difference between the two situations on the map. Blue lines will usually signify that the player must have a particular object or piece of information before he can move in a given direction (which means checking the object array). Finding hidden items requires the player to *search* for them (which requires the use of the relevant *commands*). Thus different programming requirements get different colour codes.

Chapter Five

Interior Decor - Arrays and Things

Once the basic map has been prepared it will still, despite the pretty colours, look rather bare. What it needs now is a little 'tarting up'. We have to decide what should, and should not, go into all those rooms.

The contents of the rooms will fall into two categories - things that exist in one room only and may not be moved, and things that are 'transportable'. The items in the first group need not concern us at the moment. But for the items in the second category there is a simple rule: if it exists, then it must exist *somewhere*. The somewhere is, as with the character qualities we dealt with in Chapter 3, in an array.

The process of interior decoration is, of course, another crucial part of preparing an adventure. It is also one of the easiest things to get wrong. To a certain extent, therefore, this chapter is more concerned with what *not* to do.

Little boxes ...

Before dealing with the interior decoration of an adventure in detail I want to spend a few moments discussing arrays and their uses. If you are already familiar with their function then you may wish to move straight on to the next section.

The primary purpose of an array, or a 'matrix' as it used to be called, is to store *lists* of numerical or alphabetical information as sub-units of a single variable. If we start with a 'one-dimensional' array named A$() then the various values of A$ will be labelled A$(1), A$(2), A$(3), etc. In Fig. 5.1 you will see that it holds a list of five names. So in order to create the A$() array prior to its use we would use the statement.

```
DIM A$(5)
```

A$(0)	UNUSED
A$(1)	JOHN SMITH
A$(2)	PAUL JONES
A$(3)	ERIC O'SOCK
A$(4)	BEOWULF
A$(5)	RED SHIFT

Fig. 5.1. One-dimensional array for A$().

In this line the number 5 does not refer to the *dimension* of the array but to the number of 'elements' or boxes within the array. In order to create a multi-dimensional array we must add more numbers to the DIM statement, because the 'default' dimension value for an array – the value the computer gives to the array unless told otherwise – is 1. For a two-dimensional array, for example, we would enter

```
DIM A$(5,2)
```

which would give us 5 *rows* with two *columns* in each row (actually we get *six* rows and *three* columns – see the discussion of 'zero elements' below). The number of elements in this array would be the number of rows multiplied by the number of columns, in this case 10 (see Fig. 5.2). Some machines – the Lynx, for instance – only allow

		COLUMNS				
	A$(0,0)	UNUSED	A$(0,1)	UNUSED	A$(0,2)	UNUSED
	A$(1,0)	UNUSED	A$(1,1)	J. SMITH	A$(1,2)	COLLEGE RD
	A$(2,0)	UNUSED	A$(2,1)	P. JONES	A$(2,2)	ANY ST
ROWS	A$(3,0)	UNUSED	A$(3,1)	E. O'SOCK	A$(3,2)	LAUNDRY AVE
	A$(4,0)	UNUSED	A$(4,1)	BEOWULF	A$(4,2)	VALHALLA
	A$(5,0)	UNUSED	A$(5,1)	RED SHIFT	A$(5,2)	CRAB NEB.

Fig. 5.2. Two-dimensional array for A$(). Note difference in number of 'zero' elements.

one-dimensional arrays, while others allow truly labyrinthine arrays with up to 88 dimensions! Such complicated arrays actually have very little value except when used in complex mathematical calculations, and in high-powered databases and spreadsheets. It is

very unlikely that an adventure game would ever need more than one or two dimensions in each array.

One last point. It's always a good idea to bear in mind the existence of the *zero* element in an array. It is still possible to find machines which start arrays at element '1', but most computers, including the C64, start arrays from 0. Just how big a difference the 'zero elements' make can be seen in the two diagrams above. In the C64, numerical arrays take up five bytes per element (including the actual value for the element) while string arrays take up 3 bytes plus 1 byte for each character in the string. At that rate the one-dimensional array below would take up a total of 65 bytes including 3 which are unused - less than 5 per cent wastage. In the two-dimensional array a total of 139 bytes are needed, of which 24 are unused - about 17 per cent wastage! So don't waste space unnecessarily by creating a separate array for every list if you can possibly use the same arrays for several different routines.

Don't make any array bigger than it needs to be. I have said in the line notes in the last chapter that arrays of less than 11 *positive* elements don't need to be DIMmed. At the same time, if you address an un-DIMmed array the computer automatically sets aside space for 11 elements (including the zero element). So if you really only want four positive elements, and you're short on space, then *do* DIM the array before using it.

The C64 assigns 0 (zero) as the lowest value in an array, so why not try to use the zero elements? This may not always be possible - it cannot be done, for instance, if the elements are called by variables with values greater than zero, as when printing room descriptions, e.g.:

```
IF RN > 0 THEN PRINT A$(RN)
```

All the same, it will save a lot of space if you can.

Alternatively, if you have a fairly large two-dimensional array in which the zero column is not used, then why not treat it as a separate one-dimensional array?

Program 5.1: Making the most of your zeros

The program below is a short test of the array-making process. It demonstrates quite graphically the amount of space that would be wasted by ignoring the zero elements of quite a small array.

```
]LIST

10  DIM A$(12,2)
12  FOR X = 0 TO 12
14  FOR Y = 0 TO 2
20 A$(X,Y) = "CAT"
30  PRINT X"/"Y" "A$(X,Y)"    ";
40  NEXT
45  PRINT
50  NEXT
```

Program 5.1.

And here is the printout this program produces

```
0/0 CAT    0/1 CAT    0/2 CAT
1/0 CAT    1/1 CAT    1/2 CAT
2/0 CAT    2/1 CAT    2/2 CAT
3/0 CAT    3/1 CAT    3/2 CAT
4/0 CAT    4/1 CAT    4/2 CAT
5/0 CAT    5/1 CAT    5/2 CAT
6/0 CAT    6/1 CAT    6/2 CAT
7/0 CAT    7/1 CAT    7/2 CAT
8/0 CAT    8/1 CAT    8/2 CAT
9/0 CAT    9/1 CAT    9/2 CAT
10/0 CAT   10/1 CAT   10/2 CAT
11/0 CAT   11/1 CAT   11/2 CAT
12/0 CAT   12/1 CAT   12/2 CAT
```

Think character

In Chapter 3 I spent a good deal of time explaining how to create 'progressive' characters – characters which would be modified by their experiences in the course of an adventure. On the assumption that many writers will appreciate the value of the progressive factor, and want to include it in their own games, the first rule of interior decor must be to include at least one *unavoidable* test of each character quality.

To do the subject justice, then, let's deal with all eight of the qualities included in the programs in Chapter 3 – Strength, Health, Intelligence, Skill, Height, Weight, Wealth and Luck.

Strength

As I said earlier, strength is the one character quality which should

really be included in every adventure. The original purpose of the strength rating – in board game adventures – was to help determine the outcome of a fight. Your strength rating would be used to calculate the *effect* of each hit you scored on your opponent (or opponents). Because of the time taken to calculate the effect of each blow, in a fight, few computer adventure games use the strength rating in this way. This is not to say, however, that it has lost any of its importance.

The most important function of the strength rating is, or at least should be, in helping to calculate what objects the player may carry at any particular stage in the game. In other words, the strength rating should act as a direct control over the size of the player's 'inventory'.

'And just what might an inventory be?' asks the voice in the background.

Those of you with some experience of adventuring will no doubt be familiar with the INVENTORY function under one name or another (INV, I, etc.). Basically it controls, records, and lists on request all the items that a player is carrying at any given moment. As such it is a very useful function, and saves you the trouble of constantly updating a handwritten list of objects.

Unfortunately, many adventure writers in the past have mishandled the inventory routine by using the *number* of objects carried as a way of limiting the size of the inventory. This can lead to some quite ridiculous situations. It could, for example, result in a player being able to pick up (or GET) a sledgehammer (*one* item) but have to drop something in order to pick up a box of matches and a piece of paper (*two* items) because the player is already carrying five items and may not carry more than six items at any one time. In one game that I came across – written by an amateur – this kind of programming allowed the player to carry a maximum of four items. Thus you could carry an iron stove, a deflated weather balloon, a large (i.e. man-size) wicker basket and an inflatable rubber raft all at the same time. You could not, however, carry a pair of flippers, a snorkel, a book, a fish *and* a box of matches!

Now you are going to need an inventory routine of some sort in *any* adventure game. There is no good reason, then, why each *portable* item should not be given a realistic weight to be used in calculating how many items a player may carry at any one time. The actual programming requirement is virtually the same as when inventory size is calculated by number of items (see below), though you will need an additional array to hold the weight value for each item.

One word of warning if you use this method. Don't leave too many small (i.e. lightweight) objects lying around, or stronger characters could end up with an inventory list that fills up the best part of a screen.

Program 5.2: Picking up the pieces

This first, very simple routine creates the most basic kind of inventory, taking no account of the player's STRENGTH rating or the weight of the objects themselves.

```
]LIST

1  REM *****  GET OBJECT #1  *****
2 :
3 :
8  REM *** SET UP DEMO
9 :
10  PRINT "[CLEAR]": GOSUB 1000
20 T = 0
30  INPUT "[DOWN]WHICH ROOM ";RN$
40 RN =  VAL (RN$): IF RN < 1 OR RN >
     8 THEN 30
97 :
98  REM *** 'PARSE' COMMAND INPUT
99 :
100  INPUT "[DOWN * 4]WHAT NOW ";CO$
110  IF  LEFT$ (CO$,1) = "I" THEN  GOSUB
     300: GOTO 20
120  IF  LEFT$ (CO$,1) = "G" THEN  GOSUB
     400: GOTO 20
130  IF  LEFT$ (CO$,1) = "Q" THEN  GOTO
     900
140  PRINT "[DOWN]SORRY - I DON'T UND
     ERSTAND "CO$: GOTO 20
297 :
298  REM *** DISPLAY INVENTORY
299 :
300  FOR X = 1 TO 8
310  IF OB$(X,2) = "-1" THEN  PRINT O
     B$(X,1)
320  NEXT
330  RETURN
397 :
```

```
398  REM *** 'GET' OBJECT 1 - FIND NA
     ME
399 :
400  FOR X =  LEN (CO$) TO 1 STEP
     1
410  IF  MID$ (CO$,X,1) = " " THEN N$
      =  RIGHT$ (CO$,T):X = 1
420 T = T + 1
430  NEXT
497 :
498  REM *** PART 2 - FIND THE OBJECT

499 :
500  FOR X = 1 TO 8
510  IF N$ = OB$(X,1) THEN CH = X:X =
     8: NEXT : GOTO 600
520  NEXT
530  PRINT "[DOWN]I SEE NO "N$" HERE.
     "
540  RETURN
597 :
598  REM *** TRY TO GET OBJECT
599 :
600  IF IN > 5 THEN  PRINT "[DOWN]SOR
     RY - YOU CAN ONLY CARRY SIX ITEM
     S.": RETURN
610  IF OB$(CH,2) = "-1" THEN  PRINT
     "[DOWN]YOU ALREADY HAVE THE "N$"
     !": RETURN
620  IF OB$(CH,2) = "0" THEN  PRINT "
     [DOWN]SORRY - "N$" ISN'T AVAILAB
     LE!": RETURN
630  IF  VAL (OB$(CH,2)) <  > RN THEN
      PRINT "[DOWN]THE "N$" ISN'T HER
     E!": RETURN
640  PRINT "[DOWN]O.K. - YOU HAVE THE
      "N$
650 OB$(CH,2) = "-1":IN = IN + 1: RETURN

897 :
898  REM *** QUIT ROUTINE
899 :
900  PRINT "[CLEAR]": END
997 :
998  REM *** ARRAY - FOR DEMO ONLY
```

```
999 :
1000  FOR X = 1 TO 8
1010  READ OB$(X,1),OB$(X,2)
1020  NEXT
1030  RETURN
1999 :
2000  DATA KNIFE,1,BANANAS,2,CARROTS,
      3,BOTTLE,4
2010  DATA GUN,5,PAPER,6,RHUBARB,7,WA
      TER,8
```

Program 5.2.

Line-by-line analysis

Line 10: Having cleared the screen, program execution passes to lines 1000–1030 which fills a mini-Object Array with items and the number of the room in which they can be found.

Line 20: The variable T is a supplementary value in the routine which reads CO$ in lines 400–430. As long as it precedes that loop its value can be reset to 0 either here or immediately before the loop is executed.

Lines 30–40: As this is a demonstration, the current value for RN – the player's location – must be set before each trial. See the DATA in lines 2000–2010 to find out what is in each room.

Lines 100–140: The routine will only respond to three commands: I (for INVENTORY), G (for GET), and Q (for QUIT), though it will respond to any form of these commands as long as they start with the right letter (read by LEFT$(CO$,1)). The first two commands will allow further trials; Q just quits!

Lines 300–330: Displays the inventory, if you've actually collected anything, by searching for −1 in the third element on each row of the object array. If it finds −1 it prints the name in the second element of that row (see line 650).

Lines 400–430: If told to G, or GET, an object this section of the routine moves *backwards* through the command INPUT until it finds a blank space. It then assumes that what it has collected (N$=RIGHT$(CO$,T) – which does not include the blank space) is a valid noun. This method of collecting words ignores everything between the first and last words in the command and will respond in exactly the same way to G KNIFE, GET KNIFE or even GET ME THE SHEATH KNIFE.

Lines 500–540: The program now checks the second element (remember the zero elements!) of each row for a word to match N$. If it finds a match then it sets CH to the value for that row and moves on to the next section (see Chapter 9 for more details). If no match is found, the appropriate message is generated (line 530) and the program RETURNs to be directed back to line 20 for a new command.

Lines 600–650: Because the inventory is controlled by number of items held, rather than total weight, line 600 simply checks whether we are already holding 6 items (it would only allow us past this point if IN equalled 5 or less). If we already have six items then line 600 RETURNs the program for a new command. Line 610 handles requests for items already held. Line 620 deals with items no longer available – a factor not actually used in this program but one which would apply to, say, a smashed bottle, spilt water, etc. Line 630 responds to requests for an item in another room, and 640 tells you if you have successfully obtained the required object. Only then would the program pass on to modify the third element of the appropriate row – a −1 means you are now holding that object – and the value of IN is raised by 1.

Line 900: As I said – Q quits.

Lines 1000–1030: A simple READ loop to fill the array OB$( ) with the DATA in lines 2000–2010.

Program 5.3: 'You can't carry anything that big ...'

This routine assigns a maximum weight value that can be carried by the player, and individual weight values to each item. If the player tries to pick up an item that will take the weight of his inventory over the maximum value, the message 'SORRY – YOU CAN'T CARRY ANYTHING THAT BIG' is printed out.

If you were using this routine in a program which included detailed characteristics for the player, you could obviously replace the figure 20 in line 600 with the player's current strength rating, or a formula based on that rating.

```
]LIST

1  REM *****  GET OBJECT #2  *****
2 :
```

```
8  REM *** SET UP DEMO
9 :
10  PRINT "[CLEAR]": GOSUB 1000
20 T = 0
30  INPUT "[DOWN]WHICH ROOM ";RN$
40 RN =  VAL (RN$): IF RN < 1 OR RN
    8 THEN 30
97 :
98  REM *** 'PARSE' COMMAND INPUT
99 :
100  INPUT "[DOWN * 4]WHAT NOW ";CO$
110  IF  LEFT$ (CO$,1) = "I" THEN  GOSUB
    300: GOTO 20
120  IF  LEFT$ (CO$,1)   "G" THEN  GOSUB
    400: GOTO 20
130  IF  LEFT$ (CO$,1)   "Q" THEN  GOTO
    900
140  PRINT "[DOWN]SORRY - I DON'T UND
    ERSTAND "CO$: GOTO 20
297 :
298  REM *** DISPLAY INVENTORY
299 :
200  PRINT "[DOWN * 2]YOU ARE CARRYIN
    G: "
310  FOR X = 1 TO 8
320  IF OB$(X,1) = "-1" THEN  PRINT "
    THE ";OB$(X,0)
330  NEXT
340  PRINT "A TOTAL WEIGHT OF "IN" LB
    S."
350  RETURN
397 :
398  REM *** 'GET' OBJECT 1   FIND NA
    ME
399 :
400  FOR X    LEN (CO$) TO 1 STEP
    1
410  IF  MID$ (CO$,X,1) = " " THEN N$
     =  RIGHT$ (CO$,T):X = 1
420 T = T + 1
430  NEXT
497 :
498  REM *** PART 2   FIND THE OBJECT
```

```
499 :
500  FOR X   1 TO 8
510  IF N$ = OB$(X,0) THEN CH   X:X
     8: NEXT : GOTO 600
520  NEXT
530  PRINT "[DOWN]I SEE NO "N$" HERE.
     "
540  RETURN
597 :
598  REM *** TRY TO GET OBJECT
599 :
600  IF IN +  VAL (OB$(CH,2)) > 20 THEN
      PRINT "[DOWN]SORRY - YOU CAN'T
     CARRY ANYTHING THAT BIG.": RETURN

610  IF OB$(CH,1) = "-1" THEN  PRINT
     "[DOWN]YOU ALREADY HAVE THE "N$"
     !": RETURN
620  IF OB$(CH,1) = "0" THEN  PRINT "
     [DOWN]SORRY - "N$" ISN'T AVAILAB
     LE!": RETURN
630  IF  VAL (OB$(CH,1)) <  > RN THEN
      PRINT "[DOWN]THE "N$" ISN'T HER
     E!": RETURN
640  PRINT "[DOWN]O.K.   YOU HAVE THE
      "N$
650 OB$(CH,1) = "-1":IN   IN +  VAL (
     OB$(CH,2)): RETURN
897 :
898  REM *** QUIT ROUTINE
899 :
900  PRINT "[CLEAR]": END
997 :
998  REM *** ARRAY - FOR DEMO ONLY
999 :
1000  FOR X = 1 TO 8
1010  READ OB$(X,0),OB$(X,1),OB$(X,2)

1020  NEXT
1030  RETURN
1999 :
2000  DATA KNIFE,1,1,BANANAS,2,2,CARR
     OTS,3,2,BOTTLE,4,1
2010  DATA GUN,5,3,PAPER,6,1,RHUBARB,
     7,2,BARREL,8,15
```

Program 5.3.

Line-by-line analysis

Lines 10–540: These lines are a direct copy of the lines in the last program except as detailed below.

Line 300: Just to show that this *is* the inventory a short introduction has been added.

Line 320: The program now utilises the zero elements for all rows from 1 to 8. Thus element (X,1) of the last program is now (X,0) – the name; element (X,2) is now (X,1) – the location of the object; and (X,2) holds the weight of each object.

Line 340: As each item is collected its weight is added to IN so that this line is able to display the total weight of the current inventory.

Lines 600–650: Again a fairly close version of the lines in the previous program with the exception of line 600, which now looks for a maximum *weight* of 20 lbs *including* the object to be got! I feel that the message generated by this line is a lot more satisfactory than the one shown in the last program. Line 650 is now altered to add the weight in OB$(CH,2) – translated to a number by the VAL ( ) command – to IN

Lines 900 and 1000–2010: Again as before, but now three 'values' are READ into each row of the OB$( ) array – a name, a location and a weight.

Program 5.4: Dropping out

Having discovered how to pick things up we also need to be able to put them down again. In most adventures this is done with the command DROP (item), and in this example I've stuck to that simple format.

```
LIST

1  REM *****  DROP OBJECT  *****
2  :
3  :
4  REM *** THIS SUBROUTINE ASSUMES TH
     AT
7  REM       CO$ = "DROP THE KNIFE"
8  REM       N$(NP) = "THE KNIFE"
9  :
10  FOR X = 1 TO F1
```

```
20  IF OB$(X,0) = N$(NP) THEN TE = X:
    X = F1: NEXT : GOTO 100
30  NEXT
40  PRINT "[DOWN]SORRY - CAN'T FIND "
    N$(NP): RETURN
97  :
98  REM *** IF N$(NP) EXISTS CHECK LO
    CATION
99  :
100 Q = 0
110  IF  VAL (OB$(TE,1))       PL THEN
    Q = 1
120  IF Q THEN Q = 0: PRINT "[DOWN]YO
    U CAN'T DROP WHAT YOU DON'T HAVE
    !!": RETURN
130  PRINT "[DOWN]O.K. ":OB$(TE,1)
     STR$ (PL): RETURN
```

Program 5.4.

Line-by-line analysis

Lines 10–40: A simple X loop (adventure programs can often seem to be little more than an unending series of loops!). The variable F1 is simply the highest *row* value for the object array - OB$(). We now search through the object array for the noun given in the command INPUT - CO$. If we find N$(NP), then the routine continues at line 100. If we don't find it then we politely tell the player what an idiot he is and go back for another command.

Lines 100–130: The variable Q is necessary here because the statements in lines 110–120, although they all deal with the same situation, won't fit into one program line on the C64. Setting Q to 0 and then altering its value, if necessary, is the quickest way round the problem. In line 130 the player's current location - PL - is changed from a numerical or 'real' value into a string value so that it can be entered into the OB$() array.

By the way, if you're using an inventory size controller then an appropriate statement must be added before the RETURN in line 130, either IN = IN − 1 or IN = IN − VAL (OB$(TE,2)).

Strength is not enough ...

Health

The health character quality keeps track of the player's physical

condition. If the hero catches malaria, loses a fight, etc., his health rating will drop. In the original board game this meant two things. If the health rating of any player dropped below a certain level he *had* to take a rest for a certain number of goes in order to recuperate, even though his colleagues might choose to 'play on'. If his health rating ever dropped to 0 (or below!) he was automatically declared dead and had to bow out of the game.

In computer adventure games 'resting' would mean waiting for a certain amount of time while absolutely nothing happened, a very boring situation for the player. For this reason many games use the player's strength rating to indicate his state of health, and only consider whether the player is alive (that is, whether he has a positive strength rating) or dead (if the strength rating falls below 1). Depending upon what actually happens in your adventure you may want to ignore the health quality altogether.

Intelligence

Like health, the intelligence rating comes towards the bottom of the list of useful character qualities. In the randomised character creation program in Chapter 3 I used intelligence as one of the guidelines for deciding what *type* of character the player was given. I assumed that thieves and wizards generally had higher intelligence ratings than warriors (who kept their brains in their biceps) and delvers (who kept theirs somewhere else).

Generally speaking intelligence is used to decide whether a character can *learn* new skills – languages, code-breaking methods, etc. – and should be applied to interesting problems rather than essential problems. Acquiring a high intelligence rating might make life easier for the player, but not too easy. As I said before, anyone who writes games that only a genius can complete is going to have a very small audience indeed.

Skill

The 'skill factor' plays a major role in adventure board games since they usually involve fight sequences of some kind. In that context strength would determine how *hard* you hit your opponent, and your skill rating would determine how well the blow was aimed.

In computer adventures, use of a skill rating will depend upon whether the hero is actually required to show any physical dexterity. If he uses a gun, for example, his skill rating will show how good a shot he is. Thus a person with a low skill rating might need to take two, or even three, shots before he can be *sure* of hitting his target.

(You might even include a short arcade-style test of skill before the adventure begins, so as to give the player a truly representative skill rating.)

Height and Weight

These two ratings usually go together. Their main purposes are to decide (a) whether a player is big enough to handle certain objects (very few leprechauns are seen wielding five-foot broadswords), and (b) whether the player can move through certain areas. For instance, a character who weighs 200 lbs will obviously be far more at risk when crossing ice, damaged bridges, etc., than a player who weighs only 140 lbs. And a tall, well-built character will find it much harder to avoid trouble by hiding than a 110 lb midget.

If you are short of space then height and weight, like health, are usually the first qualities to be dropped from the list of ratings.

Wealth

Apart from strength, the wealth rating is probably the most important factor in many adventures. Not only is the player's wealth very often the measure of his or her success, but it is often necessary – quite rightly – for the player to acquire a reasonable amount of money to buy the items needed to complete the game.

Luck

I've argued, in an earlier chapter, that a good adventure game is one that has *fairness* built into it. Too many games in the past have included sections where the player's success is based on sheer chance rather than skill – a complaint to be seen in many reviews. So it may seem strange to argue that a luck rating should be included as a basic part of an adventure game.

Luck, or what looks like luck, plays an important part in our everyday lives. It can be good or bad, and when players complain about the chance element in a game they usually mean that they seemed to be having an unfair share of *bad* luck. Broadly speaking, the luck rating should be used in a game to give the player a chance to win through in a situation that might otherwise seem pretty hopeless. Handled in this way, and used in moderation, the luck factor adds spice to a game rather than spoiling it.

Will the hero notice a small trap-door in a dark corner? Will he be wounded seriously, only a little, or even escape unharmed from a cunningly placed booby-trap? Will a small or poorly-armed character strike a lucky blow and defeat a superior foe? The luck

factor can be made to maintain the balance of a game, so don't be in too much of a hurry to cross it off the list of useful character qualities.

What? Where? And how many?

There's always a temptation, especially when you're working with a very limited number of rooms, to try to fill every nook and cranny with people, creatures and objects as a way of holding the player's interest. Resist this temptation at all costs!

No writer wants to waste space, and if there are too *few* items dotted around the map then the game can indeed lose its sparkle. But this is one of those occasions where too much can be as bad as, or even worse than, too little – for several reasons.

First there is the element of surprise. Let's suppose that you have decided to put two 'things' in every room, be they people, creatures, or objects. What you have done here is to give a pattern to your game, and even the slowest player will soon realise that there are going to be two things in every room. So how do you hide anything? How do you create an element of surprise? Assuming that you've used each thing for a specific purpose you'll either have to introduce a few extra things – so that some rooms contain three items – or drop some of the items you've already listed and alter the shape of the storyline accordingly.

Adopting the second alternative can be frustrating and time-consuming. Adopting the first is likely to turn even a small adventure into something that looks like a tube train in the rush hour. Which takes us back to the usual reason for overcrowding – lack of RAM space. For the space that you have saved by limiting the number of rooms (and the number of room descriptions) will soon be taken up again by the larger object arrays and the extra subroutines that you'll need to process all the added events.

On the whole it is better to keep the number of 'things' in an adventure down to a satisfactory minimum and spread them out in what would appear to be a random manner. The appearance of the occasional empty room can be most unsettling, especially if you use one or more of the subroutines which follow.

Anyone who knows anything about Melbourne House's best-selling game *The Hobbit* will almost certainly be aware of its 'animated' characters, who move about in the adventure quite independently of the player. Amazing? A trick made possible by

machine code? In one word: no. This element of *The Hobbit* is certainly a very imaginative piece of programming, yet the actual coding involved is really very simple. These next three programs – Random Item/Chase, Random Item/Limited Move and Random Item/Place – show how you too can introduce independent objects, be they things or characters, into your own programs.

Note: As they stand none of these programs will RUN by themselves. To see them at work they must be accompanied by a command input routine (to move the player character), a set of movement codes and a means of checking what is happening to the Random Item (or Items) such as a simple PRINT RI (the variable giving the Random Item's location) after each move.

Program 5.5: Hot pursuit

In this program one item, which should be carefully placed at the start of the game, is programmed to move towards the player as he or she negotiates the adventure. However, since the item moves after *every* command, and the player may give more than one command in the same room, the player and item will not necessarily meet up in the same place each time.

```
LIST

1   REM ***** RANDOM ITEM/CHASE   *****

2  :
3  :
8   REM *** PLACE RANDOM ITEM
9  :
10 RI = 42
997  :
998   REM *** CHECK FOR RI
999  :
1000   IF RI THEN   GOSUB 3000
2996  :
2997   REM *** GET RANDOM MOVE FOR RI
2998   REM       AND CHECK VALIDITY
2999  :
3000 Q =   INT ( RND (1) * PD) + 1
3010 NM =   PEEK (RI * PD + BA + Q): IF
     NM = 0 THEN   RETURN
3016  :
```

```
3017  REM *** IF NM VALID TRY TO MOVE
      RI
3018  REM      TOWARDS PLAYER AND 'RET
      URN'
3020  IF NM   PL THEN RI = NM: GOTO X
      XX
3030  IF PL < RI AND NM   RI THEN RI
      NM: RETURN
3040  IF PL > RI AND NM   RI THEN RI
      NM: RETURN
3047 :
3048  REM *** ELSE 'RETURN' ANYWAY
3049 :
3050  RETURN
```

Program 5.5.

Line-by-line analysis

Line 10: Everyone has to be somewhere, even a Random Item – which is all that this line is meant to indicate. For the method of placing random items see Programs 5.6 and 5.7.

Line 1000: Depending on its nature, the random item may be picked up by the player or destroyed. If this is the case this line should be altered to read

```
1000 IF RI=0 THEN GOSUB 3000
```

because IF RI is 'true' (i.e. the GOSUB will be executed) for any value of RI except 0

Lines 3000–3010: Q is a short-lived 'local' variable used to hold a random number. The value of PD will be the number of possible directions of movement on your map – 4, 6, 8 or 10. In line 3010 the variable NM (New Move) temporarily holds a value to be taken from a POKEd set of movement codes. The equation for the PEEK is: (the current location of the Random Item)*(the Possible Directions for movement)+(the Base Address of the Movement Codes − PD)+(a random value between 1 and PD inclusive). What this PEEK actually gives us is either a positive number – the room that the Random Item is to move to – or zero, which means that no move in that direction is possible, in which case the program RETURNs (to get the player's next command). Perhaps this will be a little clearer if I give an example.

Let's suppose that we're using a linked squares map with only one level, which gives us four possible directions for movement. Thus we

would start by either 'initialising' PD at the start of the program, giving it a value of 4 (1 = North, 2 = South, 3 = East and 4 = West), or we can substitute the number 4 for PD in the equation (often called the 'argument' in computing) in lines 3000 and 3010.

The variables RI and Q are also dealt with quite easily. RI - the location of the Random Item - should be initialised at the start of the program as shown in line 10. Its value will then be altered from time to time whenever this routine finds a new room for it to move to. The variable Q is re-set in line 3000 every time that this routine is used. Since the value of PD, in this example, is 4 then the value for Q must be 1, 2, 3 or 4. For the sake of this illustration let's assume that the Random Item is currently in room 1 (RI = 1) and that a value of 3 (for move East) has been assigned to Q in line 3000.

Finally we come to the variable BA (the letters stand for Base Address). In this routine we need the value held as BA - which should also be initialised at the start of the game or replaced by its numerical value - in order to find the correct item in the movement code table, which has been previously POKEd into memory. But the value of BA is not, as you might imagine, the address of the first byte of the movement codes. To see why this should be so let's experiment with the argument in line 3010, taking the address of the first byte of the movement code table as 20014. Substituting numerical values for each of the variables we would get.

```
NM = PEEK (1 * 4 + 20014 + 3)
```

Now what we actually want is the value held in the third byte of the movement code table, which would be at 20016. So the value we should be reading is found by PEEK(20016). But if you work out the argument above you'll find that the line is giving us PEEK(20021) - 5 bytes away! To get the correct location for any PEEK, that is to get the correct value for BA, we must subtract the number of possible directions for movement for any room *plus* 1 from the true address of the start of the movement codes. Thus our argument *should* read

```
NM = PEEK (1 * 4 + 20009 + 3)
```

which gives us PEEK(20016) the byte and value that we really want. A full explanation of the principles involved here is given in Chapter 6.

Lines 3020–3050: If the location generated for NM is also the player's current location (PL) then the value of NM is transferred to RI (the random item moves to that location) and the program moves

off to the routine which deals with that situation. Otherwise the program checks whether (a) the random item is being moved towards the player, who is in a lower numbered 'room' (line 3030), or (b) whether the random item is following the player who has somehow slipped past into a higher numbered room. In either case NM is accepted as a satisfactory move and its value is transferred to RI.

Line 3050: If moving the random item to room NM does not bring it to the player, or at least nearer, then it must be moving away. In this case the random item is left where it is and the program RETURNs to the command input routine.

There are at least two simple modifications which could be made to this routine. Firstly, if the random item is allowed to pass through walls, etc., then change lines 3000–3010 to:

```
3000 Q = INT ( RND (1) * 2) + 1
3010 IF Q = 1 THEN NM = RI + 1
3020 IF Q = 2 THEN NM = RI - 1
```

The random item will now have a 50-50 chance of moving towards the player on each turn. Secondly, if you want the random item to *elude* the player for as long as possible, then alter lines 3020–3050 as follows:

```
3020 IF NM = PL THEN RETURN
3030 IF PL < RI AND NM < RI THEN RETURN
3040 IF PL > RI AND NM > RI THEN RETURN
3050 RI = NM: RETURN
```

The random item's location will now only be altered if moving to NM takes it *away* from the player. Be careful about using this version when the random item is important to the player's success. It could be so successful that the player never catches the item up since he has no way of knowing where it is - unless you deal with that elsewhere in your program.

Program 5.6: The watchdog

In this program an item is made to move about, but within a restricted area - which it might be guarding, for instance. The player's chance of avoiding the item depends on two factors:

(1) Whether there is an alternative route past the area in question
(2) How many rooms the item moves through.

Even if there is no alternative route for the player, these rooms should include at least one 'side room', so to speak, so that the player does have some chance of passing the item without meeting it.

```
]LIST

1  REM *****  RI - LIMITED MOVE  ****
     *
2 :
3 :
8  REM *** PLACE RI (RI AREA=40 TO 45

9 :
10 RI = 43
994 :
995  REM *** IF RI EXISTS TRY TO MOVE

996  REM       IT AFTER EACH COMMAND
997  REM       BY THE PLAYER HAS BEEN
998  REM       EXECUTED.
999 :
1000  IF RI THEN  GOSUB 3000
2997 :
2998  REM *** GET RANDOM MOVE FOR RI
2999 :
3000 Q =  INT ( RND (1) * PD) + 1
3006 :
3007  REM *** CHECK RESULT AGAINST
3008  REM       THE MOVEMENT CODES
3009 :
3010 NM =  PEEK (BA + (RI * PD) + Q)
3017 :
3018  REM *** MAKE MOVE IF VALID
3019 :
3020  IF NM = 0 THEN  RETURN
3030  IF NM < 40 OR NM > 45 THEN  RETURN

3040 RI = NM
3046 :
3047  REM  *** CHECK FOR RI/PLAYER
3048  REM       MEETING AND HANDLE
3049 :
```

```
3050  IF RI = PL THEN  GOTO XXX
3057 :
3058  REM *** OR SIMPLY 'RETURN'
3059 :
3060  RETURN
```

Program 5.6.

Line 10: As you see, I've chosen to confine the random item to the area covered by rooms 40 to 45 inclusive. In this case, therefore, it is specifically placed at the centre of this area at the start of the game.

Lines 1000–3020: These are the same as lines 1000–3010 in the last program.

Line 3030: If changing RI to the value of NM would take the item outside its alloted area then don't change it.

Lines 3040–3060: Otherwise move the random item and, if it meets the player, go to the routine which deals with that situation. If there's no meeting then RETURN for another command input.

Program 5.7: The random monster

In this third program one or more items are placed randomly at the start of each game. This last routine is in many ways the most effective, since a given room might come up empty for several games in a row, and then suddenly become inhabited by an aggressive monster in the next.

If more than one item is to be placed by this method you may wish to include the 'filter' section, which ensures that each item is definitely placed in a different room. If you're only placing one item the filter will not, of course, be necessary.

```
]LIST

1  REM *****  RI - RANDOM PLACING  **
     ***
2 :
3 :
8  REM *** SET LOOP TO NO. OF RI'S
9 :
10  FOR X   1 TO NR
17 :
18  REM *** EACH WITH RANDOM LOCATION
```

```
19 :
20 T =  INT ( RND (1) * TR) + LR
27 :
28  REM *** FILTER OUT REPEATS
29 :
30  FOR Y = 1 TO X - 1
40  IF  VAL (OB$(Y,1)) = T THEN Y = X
      - 1: NEXT Y: GOTO 20
50  NEXT Y
57 :
58  REM *** AND PLACE RI
59 :
60 OB$(X,1)     STR$ (T)
67 :
68  REM *** THEN REPEAT FOR NEXT RI
69 :
70  NEXT X
80  END
```

Program 5.7.

Line-by-line analysis

Line 10: Sets up a standard loop for 1 to NR, the total number of random items to be placed.

Line 20: Gets a random value for T that is between LR and the total number of rooms in your map. LR is the number of the Lowest numbered Room in which any random item will be placed. TR equals the Total number of Rooms *minus* LR.

Lines 30–50: One way of simplifying your control of random items is to include them in the object array. In this case I've assumed that the random items will occupy the first NR rows of OB$(). This means their names, initial locations and weights will already be in the array, and this routine will *relocate* them in a random fashion. Line 40 checks whether the location for the next item is already occupied by another *random* item, and gets a new location if it is. If you don't want any doubling up at all then make the top value for the Y loop the top *row* value for the object array. If you don't care what goes where then delete lines 30–50.

Lines 60–70: Once the routine has found a satisfactory location for item X, the numerical value of T is transformed into a string value and placed in the appropriate element of OB$() and the X loop, if still incomplete, is repeated.

The last objection to using large numbers of items in a game concerns the time needed to process each command. The best commercial games are nearly all written in machine code. There is nothing that machine code can do that BASIC cannot handle – but machine code does everything very much faster. Thus machine code programs can be made far more complicated than BASIC programs and still run in the same amount of time. If you can write your programs in machine code then RAM space alone is the limiting factor in setting a maximum size for your object arrays and event handling routines. If you're working in BASIC, however, remember that as the player enters each new room, the control program will have to scan through the entire list of objects to see which of them, if any, is in the room.

You can complicate a game by making objects and problems interrelated rather than using a larger number of 'one off' items. For example, if there is a bottle in one of the rooms near the start of the game, why not make it a multi-purpose bottle? When the bottle first appears in the game it may be hidden, as it contains a piece of paper on which is written an important clue. Then, in case the player is tempted to keep the clue and discard the bottle, it can be used to carry liquid, a small animal such as a scorpion, or some fine-grained substance like sand. Once it has served this new function it might then be used as a weapon, and since this will inevitably lead to it getting broken why not use the edge as a cutting tool? In other words, if an item *must* be included in the game for one purpose why not see if it can be used somewhere else as well?

Program 5.8: Coming or going?

There is, in fact, one way of using the same object array *twice* in the same game (though the control program will of course be longer). To use this system the game must be set up as a two-part journey – there and back. This will allow you to have one set of items in play on the outward journey, and a second set for the return trip. The one condition here is that all items used will be unique to one journey. That means you won't be able to use an item collected on the outward journey when you are on the way back. If you do need any item on both journeys it will have to be re-introduced in the second array and the player will have to GET it again, unless the second array is carefully set up so that currently held items are preserved. This can be done – as in the program below – but it isn't worth the

extra program space/execution time unless it is absolutely essential.

Tip number 6: go for quality rather than quantity. Most adventurers will gain more satisfaction from overcoming one really challenging problem than they will from resolving half a dozen problems that look petty and pointless once the solutions are known.

```
]LIST

1  REM *****  SIMPLE ARRAY FILLER  **
     ***
2 :
3 :
10  FOR X = ST TO F1
20  READ OB$(X,0),OB$(X,1),OB$(X,2)
30  NEXT
40  END
9997 :
9998  REM *** ITEMS FOR OB$() ARRAY
9999 :
10000  DATA OB1 NAME,OB1 LOCATION, OB
     1 WEIGHT,OB2 NAME,OB2 LOCATION,O
     B2 WEIGHT
10010  DATA ETC.
```

Program 5.8.

Line-by-line analysis

Lines 10–40: Although the basic array-filling routine is included in several other programs, I thought I'd include it for the use of any newcomers to computing. Line 10 sets up a simple loop by which the same operation (or operations) is/are repeated a given number of times. In this case I want to fill an array of (F1−ST) rows by 3 columns, ST being the lowest row number and F1 the highest. Because the value of X increases by 1 each time we go round the loop we are saved the bother of writing something like

```
20 READ OB$(1,0),OB$(1,1),OB$(1,2)
30 READ OB$(2,0),OB$(2.1),OB$(2,2)
```

and so on. Line 30 simply sends the computer back to the beginning of our loop (to the start of line 20, in fact) until X = F1. Incidentally, when a loop is completed the value of the loop index – in this case X – is actually 1 higher than the highest control value. So don't make the mistake of thinking that X = F1 when you get to line 40; it actually equals F1 + 1.

Lines 10000–10010: Whenever a program includes the command READ (as opposed to READ#) it looks for the earliest piece of DATA in the program that has not yet been used by another routine and proceeds to collect items (separated by the commas) until the READ command has been completed.

Once we understand the basics of filling and using arrays the job of altering them, if necessary, becomes a whole lot easier, especially if you keep a note of what is in each element of the array. This isn't too difficult if you use one of these two simple routines. Of course you will need to have used the usual system of setting your program out with all the line numbers as multiples of ten (10, 20, 30 and so on) in which case you'll have plenty of room to insert either routine, with the line numbers changed accordingly, at any point in your program where you want to check the contents of a given array. The first routine will display the contents of a one-dimensional array, the second will set out a two-dimensional array. Both routines (which will not, of course, do anything unless they are part of a larger program containing at least one array) will hold the array display on the screen until you press a key to allow program execution to continue.

```
12 FOR X = 0 TO 10

13 PRINT "ELEMENT ";X;"="A$(X)

14 NEXT X

15 GET Z$: IF Z$ = "" THEN 15
```

Note: This first routine assumes that the array we want to examine has been set up, at the start of the program, using the statement DIM A$(10), though it could, of course, be altered to read any part of an array rather than the whole by altering the values in line 12.

In this next routine we will examine an array that has been set up using the statement DIM A$(10,10) using what are called 'nested loops' (one loop inside another).

```
12 FOR X = 0 TO 10

13 FOR Y = 0 TO 10

14 PRINT "ELEMENT ";X;"/";Y;"="A$(X,Y)
```

```
15 NEXT Y

16 GET Z$: IF Z$ = "" THEN 16

17 NEXT X
```

Note: It is not necessary, on the C64, to specify the variable name in the NEXT statement of a loop, though it is advisable to do so when you are using nested loops. If you do specify your variables then make sure that you always follow the FILO principle - first in, last out - or your program, while it may not crash, will certainly not handle the work within the loops correctly.

You will also notice that the GET Z$ statement, which causes the breaks between displays, has been inserted *between* the two NEXT lines. This is necessary when examining an array of this size as it causes the program to pause after each *row* of the array has been displayed. If we tried to display the entire array in one go most of it would run straight up beyond the top of the screen before we had time to read it.

Program 5.9: Re-filling an array

The next subroutine, used to re-fill an array that has already been in use for some time, makes use of the nested loop system I described just now. In this case the outer loop is set for the number of *new* objects to be placed in the OB$() array, whilst the inner loop controls the search of the rows of OB$() as we look for 'free' array space.

```
]LIST

1  REM *****  ARRAY REFILL  *****
2 :
3 :
8  REM *** STRIP CURRENT OB$() ARRAY
9 :
10  FOR X = 1 TO F1
20  IF OB$(X,1) <  > "-1" THEN OB$(X,
     1) = "0"
30  NEXT
97 :
98  REM *** AND INSERT PART 2 OBJECTS
```

```
99 :
100  FOR X = 1 TO F2
110  FOR Y = 1 TO F1
120  IF OB$(Y,1) = "0" THEN  READ OB$
     (Y,0),OB$(Y,1),OB$(Y,2):Y = F1
130  NEXT Y
140  NEXT X
19997 :
19998  REM *** ITEMS FOR PART 2 ARRAY

19999 :
20000  DATA
20010  DATA
```

Program 5.9.

Line-by-line analysis

Line 10: This looks like any other start to a loop, but it isn't! In this context I've assumed that the player may still be carrying some items gathered in the first part of the game. And unless I've inserted some kind of control – like 'YOU CAN ONLY CARRY THREE ITEMS BEYOND THIS POINT' – I won't know exactly how many items he has. Now as everyone knows you *can't* DIM the same array twice (not unless you want to crash your program!), so I have to have made the OB$() array big enough at the start of the game to allow room for all the items to be used in the second part of the game *plus* any that the player is still carrying. F1 then, is the *maximum* size of the OB$() array.

Lines 20–30: The rest of the loop runs all the way through OB$() and sets the location element of any item the player isn't carrying to zero.

Lines 100–140: I can now insert all the items for the second part of the game (a total of F2 objects) into the 'empty' spaces in OB$(). The X loop deals with all the part two items. The Y loop searches OB$() until it finds a vacant space. The statement Y = F1 in line 120 makes sure that I don't waste time looking for more vacant spaces when I've just filled one. Nor can any items be unintentionally carried over from part 1 to part 2: by the system we've been using up until now, if OB$(X,1) = "0" then that item is automatically unavailable.

Now you have it, now you don't

The last routine I want to deal with before going on to make some general comments is the SAVE GAME option.

This option is designed to allow players to SAVE a game while it is in progress, either to preserve a situation before moving into an unknown area (so they can start again at the last room successfully dealt with, rather than having to go right back to the beginning of the game), or so that they may end that session and come back later to pick up the game where they left off. Many players think this option is essential, but it has yet to become a standard feature of all adventures.

Quite why so many writers fail to include this option I really don't know, since it involves nothing more than saving the current room number plus the two arrays which hold the character status and the list of objects. This is achieved with a couple of simple loops which store the information in a sequential file called SAVED GAME, or whatever. After the game itself has been loaded, then, the player is asked whether he wants to (1) play a new game, or (2) recommence an old game. If he chooses option (2) then the information is read back into the computer so that it replaces the DATA set up at the start of each game. The means of saving simple and array variables is clearly laid out in the manuals for the cassette player and the disk drives. The only point of deviation from these procedures would occur if some information – such as the character status array and name – had been POKEd into memory. In this case it would be necessary to reassign the relevant values to a set of variables before saving them.

Mr Nice and Mr Nasty

Two of the most common faults in adventure games come about when the writers are either too helpful or too devious. There is, in fact, a very thin dividing line between being fair and being obvious. Being fair means constructing an adventure so that it follows a consistent set of rules, and setting problems that anyone with a reasonable amount of intelligence and general knowledge should be able to solve – given enough time.

Being obvious means ... well, let me show you what I mean with an example from a game I came across in a magazine a few months ago. In this game the only correct solution to the problem that provided the climax of the game was to drop a banana skin at the right moment. This skin was placed, when the game began, in a room about halfway through the map, and in that position seemed to serve no useful purpose whatever. Moreover the room containing

the skin was on one of two possible paths through the same area.

So far, so good. Unfortunately the writer ruined what would have been a rather clever and unusual piece of problem-setting by making it impossible to leave *any* room containing the banana skin unless the player was carrying the skin. Thus there was absolutely no need for the player to work out the purpose of the skin in advance, since he had to be carrying it when he came to the final problem!

Tip number 7: don't underestimate the player's ability by telegraphing the value of clues, objects and characters. Exploration of uncharted worlds is, after all, an important part of the lure of adventuring.

While the over-obvious program can be irritating, an even bigger problem, and one which tends to occur more frequently, arises when adventure writers try to be too subtle. Take this example, drawn from a commercially available game.

At the very start of the game the player finds himself in a store-room containing just three objects. He is allowed to GET any or all of these objects, though none of them has any immediate value. If he uses the INVENTORY command before leaving this room then he will only be told which of those three objects, if any, he is carrying.

The player now proceeds to room 2 (room 1 has only one exit) and finds himself engulfed by total darkness. So what does he do? He cannot use any of the objects from room 1 to lighten his path – maybe he should just keep on moving. But the result of that decision is instant death! In fact the writers have created a situation where the only correct choice is to call up the inventory again – though there is no earthly reason why anyone would. If you do ask to see the inventory, lo and behold – you're carrying a lamp!

Now where this lamp comes from is anyone's guess. It certainly isn't hidden in room 1, and you can't GET it in room 2. It may well be that the writer(s) of the game have some logical explanation for this incident, but I suspect that they alone know what it is.

Tip number 8: be logical. There's no reason why your game shouldn't have some very strange twists to the plot – so long as the player has a *genuine* chance of finding out what's going on and how he can deal with it. (And I don't mean that every game should have a book of 'hints and answers' like the one supplied with the game I've just quoted. This solution is, all too often, a substitute for good planning.)

Skinning the cat

The old saying that 'there's more than one way to skin a cat' may not, at first sight, seem to have much to do with adventure games. Yet it leads us to the next common fault made by adventure writers – leaving too many opportunities open. In the opinion of some writers (and reviewers) every item in an adventure should have a use. This view has its value, but it also has its faults. If every item has a purpose then the adventurer will need to collect every item at some point in the game. This leads to moments when the player will need to make strategic decisions about what to hold on to and what to drop. And God help you if you make the wrong decision!

On the other hand, if the player knows that he must collect each item he will soon be able to compile a list of what's available, and work out the solution to each problem on the basis of logic rather than by applying truly creative thinking. Of course we could give every item a purpose – but make it a negative purpose in some cases. Thus a packet of flash powder (as used by magicians) could be left lying around close to a room with an open fire. Pick up the flash powder and go too close to the fire and the powder erupts, giving you a very nasty burn or blinding you temporarily.

Personally I prefer the third alternative – positive and negative articles plus red herrings. This last group not only serve to confuse the player (do I really need a left-handed widget?) but also serve as problems in themselves (what do I do with a burnt-out light bulb and a broken tripod?). Moreover since some articles are immovable (see below) a carefully set-up situation can have a player spending a confused half-hour or so trying to work out, for instance, how to shift a tin bath bolted to the floor of the outhouse.

Some of the best red herrings I've come across actually resemble items in the same game which have real value. Thus you may find a truly sturdy-looking broadsword that only reveals its weakness when the player tries to use it in battle (it has been cut half-through close to the hilt and the hole filled with dirt – you can guess what happens when it strikes another sword).

The last group of objects to be included in a game – though they are *not* stored in the 'object array' – are the immovables. These are objects which always remain in the same room and can be useful, harmful and red herrings. The purpose of this group of objects is to provide extra items without using up much extra space. Because they cannot move, or be moved, their weight and location is irrelevant. Moreover, since it is generally assumed that being in the

same room as an immovable will inevitably bring you into contact with it, we don't even need to store a name for an immovable except in the room description. To deal with attempts to GET an immovable object (assuming that you don't include too many of them) first let the program check the object array. If it can't find the name of the required article then check which room the person is in, and tell them the object can't be got if it is one of the relevant rooms.

Which brings us neatly back to the first subject of this section – cat skinning. The point is that one object can often have several uses, as in the case of the bottle I mentioned earlier. And if one object can be used several ways, so one purpose can be served by several objects. Take cutting a rope as an obvious example. It could be done with a knife, a piece of glass, a sharp bit of rock or even, if you're prepared to run the risk, by setting fire to the rope. That's a fairly obvious example, and not one a writer is likely to overlook when they're setting problems. Yet the same *kind* of mistake still crops up regularly in commercial games in one guise or another.

There is one very well known game, for instance, where the player should conceal himself in a barrel at one point in the story. The trouble is that there are several barrels stored in the same place. If the player reaches the right room before one of the barrels is removed then, so long as he makes the right choice (that is, to hide), the program automatically assumes that he is hiding in the right barrel. If the player arrives *after* one barrel has been removed and tries to hide he will be told this is not possible. Which is none too smart, as there are still plenty of empty barrels lying about!

This is an illogical situation, the result of insufficient thought. If the program had been capable of dealing with just one more item – the placing of the only correct barrel – then there would be no problem. The other hiding places could be left open and the 'HIDE' command would be obeyed (but useless). Alternatively all the other barrels could be sealed off. Either way the whole situation *could* have been dealt with quite easily, and remained consistent.

Tip number 9: double check. If the only acceptable way out of a given situation is by breaking the window, and if the window must be broken with a stone, then make sure that the glass is described as being too thick to be broken by a bare fist. And make sure that the player has no other implements at hand with which to do the job. In other words, take time over the problems you set, and if you do find alternative solutions to a problem either block off the ones you don't want or expand the program to deal with them. Whatever you do, don't just brush them aside and hope they won't be noticed.

Program 5.10: The unexpected

You bend down to pick up a coin – and the ceiling falls in on you. You walk into an empty room – and drop thirty feet because the floor was an optical illusion. You find a shortcut – and get skewered by a set of bamboo spikes. Yes, my friends, these are the booby-traps. Beloved by adventure writers, feared and hated by adventure players, booby-traps are themselves a kind of booby-trap.

Like most things, booby-traps can be used or abused. They come in two main varieties: those which can be made safe (defused) and those which can only be avoided by sheer luck or well-thought-out tactics.

Type one traps can occur anywhere. They may be set to penalise the careless player, to ensure that the player is carrying a certain object (an object with more than one purpose), or simply to add an element of danger. They are also a legitimate form of problem-setting in their own right, of course, and finding devious solutions to tortuous situations can be as much fun for the writer as it is for the players.

Type two traps, which are almost always fatal, differ from type one traps in a quite important way. Firstly, because they cannot be defused, they should *not* be placed on a route that the player *must* take. Their main purposes are to prevent players taking short cuts, to penalise the over-daring and greedy player, and to guard objects or clues that aren't essential to the game. The one qualification to this last point is that a room containing an important item or clue may have one entrance/exit which is guarded by a defusable booby-trap, and another which is sealed by a non-defusable booby-trap so that the player must go out the same way that he came in.

Tip number 10: use booby-traps harshly if you will, but be fair. The routine below is a typical example of a 'defusable booby-trap'. It was invented to fit a situation where the player is required to open a safe with a combination lock – provided that he has collected the numbers elsewhere in the game. Since it is always possible that a player may, in his excitement, hit the odd wrong key, this routine generously allows ten digits to be entered, even though it only requires three. In other words, you're allowed seven wrong digits. Hit an eighth *wrong* key, however, and it'll be the last thing you do in this adventure.

```
]LIST

1  REM *****  RANDOM LOCK  *****
8  REM *** SET RND VALUE FOR LOCK
9 :
10 LO =  INT ( RND (1) * 999) + 1: IF
     LO < 100 THEN LO = LO + 100
20 LO$ =  STR$ (LO):LO$ =  RIGHT$ (LO
     $,3)
30 MT = 10
40  PRINT LO$
97 :
98  REM *** EXPLAIN SITUATION
99 :
100 Q = 1: PRINT "[DOWN]THE SAFE HAS
     A COMBINATION LOCK!"
107 :
108  REM *** AND ALLOW MT ATTEMPTS
109 :
110  FOR X = 1 TO MT
120  PRINT "[DOWN]PLEASE ENTER DIGIT
     NUMBER "Q" ";: INPUT TL$
130  IF TL$ <  >  MID$ (LO$,Q,1) THEN
      PRINT "[DOWN]SORRY - THAT WAS W
     RONG.": GOTO 150
140  PRINT "[DOWN]WELL DONE.":Q = Q +
     1: IF Q = 4 THEN X = MT: NEXT : GOTO
     200
150  NEXT
157 :
158  REM *** RELEASE BOOBY-TRAP
159 :
160  PRINT "[DOWN][RVS]OOPS! AFTER "(
     MT - Q + 1)" FALSE TRIES A WELL
     ";
170  PRINT "PLACED BOOBY-TRAP SENDS A
      BLOCK OF STONE";
180  PRINT "DOWN ON YOUR HEAD.      R
     .I.P.!!!": END
197 :
198  REM *** OR OPEN SAFE
199 :
200  PRINT "[DOWN]THE SAFE IS NOW OPE
     N!!!": END
```

Program 5.10.

Line-by-line analysis

Lines 10–20: For the later part of this routine I need a three-figure number, but I need it in string form. Line 10 gets a random number and line 20 converts it into a string.

Line 30: I also need a value for the Maximum number of Tries that the player is allowed before the booby-trap is activated. MT can equal any reasonable number.

Line 40: Prints out LO$ for test purposes only. This line should obviously not be included in the final version of the program.

Line 100: Q will be the controller for the number of successful attempts. It is also used as part of the instruction printout to show which digit the player is looking for and as the index to LO$ when we look for an INPUT/required number match (see lines 120–130). That's why it must be set to 1 rather than 0. The rest of the line merely sets the scene for the player.

Lines 110–150: Another loop! But quite a clever one. Line 120 looks for a digit to be input to TL$ using Q to tell the player which digit he's looking for. Line 120 compares TL$ with the *required* digit in LO$. If they don't match, an error message is generated and the program tries to execute another loop. If TL$ and MID$(LO$,Q,1) do match then brief congratulations are offered and Q is incremented by 1. If Q now equals 4 then, since we only want three digits, the player has succeeded and moves to line 200.

Lines 150–180: If, on the other hand, the player uses up ten attempts without finding the right combination then the program 'falls through' line 150 and the player is 'stoned to death' (aren't you glad it's only a game!). At this point the game does, of course, END.

By the way, if you want to adapt this routine so that even one attempt at opening the safe will trigger the booby-trap simply set MT to 1. The routine could also be altered to one that is *time* dependent. In this case line 110 would be altered to

```
110 TA = TI + (seconds allowed * 50)
```

Line 150 would be altered to

```
150 IF TI < TA THEN 120
```

Chapter Six

One Step at a Time

The processes described in this chapter and the next are, as you will soon see, rather more technical than most of the other material in this book. So if you're fairly new to computing, or haven't ventured beyond straightforward BASIC programming before, don't feel too bad if, at first reading, these chapters look impossibly complicated. Like most things in computing, playing around with the internal processes of the C64 gets much easier once you have a bit of practical experience to look back on.

Once your basic map is complete, including the placing of objects, you will need to lay out a table of 'movement codes' based on that map. Although these codes are comparatively simple to construct, and even easier to control from the main program, they are a central feature of any game and it is imperative that the movement code table is one hundred percent accurate. After all, it doesn't really help to know where you are if you don't know where you've come from or where you are going.

If you turn back to the map in Fig. 4.6 for a moment you'll see that there are three possible routes in and out of room 1 - via room 2, room 3 or room 19. Of course *you* can see what I'm talking about, because you can *see* the map. But how would the computer understand this information? It cannot literally see the map, so it must be given this knowledge by some other means. In fact it must have access to what is often called a 'look-up table', which holds the details on the map in *numerical* form. This is the only efficient way of storing the map in the computer so that it can tell which moves are valid for each room.

You'll notice I didn't say 'the only way' but 'the only *efficient* way' of storing the required information. This takes us back to the 'calculated move' routine which I described in Chapter 4.

At first sight the grid map and calculated moves system looks extremely effective. But only at first sight. For when you come to

examine the programming details more closely you find that the whole thing is a waste of time (despite the fact that at least two other books on adventure writing feature this routine as *the* key to movement control). And I'm not just referring to the error which allows you to 'fall off the edge of the world'.

To recap very briefly on the earlier discussion of this system, the basis of the calculated move grid is that each *row* of rooms within the grid is of equal length. Thus to move NORTH you deduct the number of rooms in a row from the room number of your Current Room number - CR = CR − RL. To go SOUTH you *add* row length to your current room number - CR = CR + RL. And to go WEST or EAST you deduct 1 or add 1 to your current room number respectively - CR = CR - 1 (for GO WEST) and CR = CR + 1 (for GO EAST).

On the face of it, assuming that we include the error trapping routine in Program 4.2, this all looks perfectly logical. And it is. Except that we don't need to *calculate* the new location in the first place!

Under, over, sideways, down

If we stick with squared maps for the moment it is possible to move in at least four directions - NORTH, SOUTH, EAST and WEST. Add movements UP and DOWN and you can move six ways. And we *could* calculate the result of any move. (To move UP one level the new room number will be found by adding the current room number to the total number of rooms on the current level - CR = CR + TR.) The fault in this method really only becomes clear when you look at what happens *after* the move has been calculated. Because at some time or other the writer has to use a second routine which tells the computer whether the move is legal (that is, whether the player has moved through an open doorway or through a brick wall, for example). And to do this the computer has to check ... yes, you've guessed it - a set of movement codes!

'Sounds good. But what on earth are you talking about?' says the voice in the background. 'Of course you have to check whether a move is legal - unless every possible move is legal.'

The voice is right, of course. So let me explain myself more clearly. I said just now that a square room has six possible exits. And if we use a separate variable for each direction, as we must, then we would end up with a list of movement codes which looks something

like this:

N = 0 : S = 0 : E =1 : W = 0 : U = 1 : D = 0

The logic here is pretty obvious. The variable for each direction must be given one value (above 0 or below 1) if movement is allowed, and an opposite value (below 1 or above 0) if movement is not allowed. So, all that we need to do to process each 'grid move' is (a) calculate the result of the intended move, (b) check that the move is legal, and then (c) move the player to the new location or generate 'bad move' message. It all looks very simple and straightforward. Yet this method actually involves a redundant step – stage (a). If we're going to use a 'look-up table' to check for legal moves then we might just as well make that table a record of *destinations* and illegal moves, both at the same time.

In this case a single sector of the table might look like this:

N = 0 : S = 0 : E = 19 : W = 0 : U = 101 : D = 0

Using a table laid out in this form allows us to eliminate the calculation part of each move in favour of direct collection of data.

O.K.? Well, not quite. Using the calculation method does *appear* to have the advantage of using far fewer bytes, even in its modified form. After all, the calculation routine itself takes up very little space (less than 150 bytes) and each value in the table of movement codes, or movement 'validation' codes, takes up just one byte. By adopting the second method we free much of the space used for the calculation routine (saving about 100 bytes), but at the same time we are forced to use up an additional byte for each code with a value greater than 9 – or an extra two bytes if the value is over 99. In an adventure based on just 99 rooms, with only 4 direction codes for each room (no UP or DOWN), this second method could use an extra 1000 bytes or more!

'I thought you were going to show people how to *save* space, not how to waste it,' says the voice.

And so I will. Let's look at the problem again.

Program 6.1: One byte at a time

The idea that each digit of each room number must take up one byte is based on the standard method of storing movement codes, in an array. Using this method we would indeed be wasting space, not only for each additional digit but also for each array element.

And that is assuming that the array is loaded directly into the computer's memory. There are still many programs which initially hold the movement codes in DATA statements so that the values must be READ from the program and re-stored elsewhere in the memory. It's true that the C64 will allow you to store a zero in a DATA line without actually having a digit in the line, as in:

```
DATA ,,19,,101,
```

This line would be READ as 0, 0, 19, 0, 101, 0. But even so there must be a comma (one byte) for each and every value. At this rate the movement codes for even a small adventure map eat up RAM at a terrible speed. Fortunately there are alternatives.

The simplest option is to re-value all direction variables each time you enter a new room (as in the program that follows). This way you avoid the use of an array altogether, so you don't need any array space and you don't need any array pointers. However, you will still be wasting a couple of bytes for each change of value, and you will still need to use one byte for each digit.

Please note that neither this program nor Program 6.2 – a second method of directly revaluing the room movement codes – will have any effect when RUN by themselves. They are 'how to' programs, and must be linked into a larger program containing a command parsing routine (see Programs 8.1 and 8.2 in Chapter 8) plus a set of room descriptions and movements codes.

```
]LIST

1  REM *****  DIRECT REVALUE #1  ****
     *
2 :
3 :
95  REM *** ROUTINE TO MOVE TO NEW
96  REM       LOCATION (WHEN N$ IS THE
97  REM       SECTION OF CO$ CONTAINING

98  REM       A MOVEMENT INSTRUCTION
99 :
100  IF  LEN (N$) > 1 THEN 2000
110  IF N$ = "I" OR N$ = "L" THEN 180
     0
120  IF N$ = "N" AND N > 0 THEN RN
     N: GOTO 190
130  IF N$ = "S" AND S > 0 THEN RN
     S: GOTO 190
```

```
140  IF N$ = "E" AND E > 0 THEN RN =
     E: GOTO 190
150  IF N$ = "W" AND W > 0 THEN RN =
     W: GOTO 190
160  PRINT "[DOWN]I CAN'T MOVE "N$: GOTO
     X: REM X=START OF COMMAND INPUT
     ROUTINE
170 :
190  ON RN GOSUB 10000,10050,10100
200  GOTO X: REM  X=START OF COMMAND
     INPUT ROUTINE
9997 :
9998  REM *** START OF ROOM DESCRIPTI
     ONS
9999 :
10000  PRINT "(ROOM DESCRIPTION)"
10010 N = 0:S = 11:E = 14:W = 0
10020  RETURN
```

Program 6.1.

Line-by-line analysis

Line 100: I have started this program with the assumption that most of the commands being used will be more than one letter long. The only one-letter commands that are accepted are I (Inventory), L (Look) and N, S, E, W (GO NORTH, GO SOUTH and so on). So line 100 filters out all commands longer than one letter and sends the program execution to line 2000 (not included here) to deal with them.

Line 110: This leaves us with six valid commands *and* any illegal one-letter commands entered by mistake. This line filters out commands I and L, sending the program on to the routines at line 1800 (also not included here) if it finds a match between N$ and either of these letters. This means that when we get to lines 120-160 we only have to deal with commands to move North, South, East or West, or any illegal entries.

Lines 120-150 deal with the four legal commands – N, S, E and W. A check is made to see which letter has been entered as the command, and the value of the appropriate variable is read to see if it is 0 (zero) – which means that you cannot move in that direction – or a positive number, which would show which room to move to next.

So where do we get the values of N, S, E and W from in the first place? In order to understand this we must jump forward to lines 10000–10020 for a moment.

Lines 10000-10020: Right at the start of any adventure game you must move the player to the room at the start of the map (normally room 1). You would then set PL - the player's location variable - to equal that room number (just as it is updated for each legal move in lines 120-150), display a room description, as in line 10000, and initialise the movement code variables for that room. In this example, in line 10010 the movement variables have been set so that the player can *only* move SOUTH, to room 11, or EAST, into room 14. Variables N and W are both set to 0 for 'no move allowed'.

If this was the start of a game then the program would move on to collect the player's first command input. In this example, however, the player's location (indicated by the variable PL) is probably about room 12 or 13 (PL=12 or PL=13), so instead of going straight on to the command routine we RETURN from line 10020 to line 200.

Lines 190-200: If we've managed to make a move then RN will have been set to a new number. We use this, in line 190, to direct the program to the correct set of lines to print out a room description and re-set the movement variables (as in lines 10000-10020). When the program RETURNs, to line 200, we must direct it on to another section of the program which will check what the results of this move may be, if any (see below).

Finally, we must deal with moves that *cannot* be executed, either because the required movement variable has a value of 0 or because the command is illegal. Thus, rather than having line 160 read I CAN'T MOVE THAT WAY we use a more general message: I CAN'T DO THAT. Program execution then goes back to the start of the command input routine for a new command.

Before we move on to the next program I'd like to deal very briefly with the 'move result check' that I mentioned just now. Many programs simplify the situation by not having any results from a straightforward move command other than making the move itself. In this case line 200 would send program execution back to the start of the command input routine. Whichever way you want to shape your programs the 'result check' routine would be made up of lines like this:

```
5000 IF PL = 10 THEN GOTO 7000
5010 IF PL = 14 AND OB$(5,1) = 14 THEN GOTO 8080
5020 IF PL = 26 AND OB$(12,1) <> "-1" THEN GOTO 9160
5030 IF OB$(45,1) = PL THEN GOTO 9530
5200 GOTO X
```

In other words, we are checking to see whether the conditions are right to initiate a choice of 'event routines'.

In line 5000 the event will occur, or go on occurring, as long as the player is in room 10. In line 5010 the event will only occur if both the player *and* object 5 are in a given room at the same time, object 5 being a moving Random Item (see Programs 5.5 and 5.6).

In line 5020 the event will only occur if the player moves into a given room and is *not* carrying object 12. Finally, in line 5030, the event occurs in any room if object 45 is in the same room as the player. Note that the event check routine *must* end with a line to return the program to the start of the command input routine (at line X) if none of the event conditions is met.

Program 6.2: Packing it tighter

A second method, which involves *slightly* more complex programming, saves bytes by removing the need to reset each value. Unfortunately it uses up a good proportion of the space you have saved because all values must contain the same number of digits. Thus, if your map has 99 rooms, all values below 10 will need a leading zero attached to them as in 01, 02, 03, etc. And if you have 100 or more rooms, values below 10 will have two leading zeros – 001, 002, etc. – and numbers from 10 to 99 will have one leading zero. The method of accessing movement codes stored in this manner is shown in the following program.

```
]LIST

1  REM *****  DIRECT REVALUE #2  ****

2 :
3 :
95  REM *** ROUTINE TO MOVE TO NEW
96  REM       LOCATION (WHEN N$ IS THE
97  REM       SECTION OF CO$ CONTAINING

98  REM       A MOVEMENT INSTRUCTION
99 :
100  IF  LEN (N$) > 1 THEN 2000
110  IF N$ = "I" OR N$ = "L" THEN 180
     0
120  IF N$ = "N" THEN DI = 1: GOTO 17
     0
```

```
130  IF N$    "S" THEN DI = 3: GOTO 17
     0
140  IF N$    "E" THEN DI   5: GOTO 17
     0
150  IF N$    "W" THEN DI = 7: GOTO 17
     0
160  PRINT "[DOWN]I CAN'T MOVE "N$: GOTO
     X: REM X=START OF COMMAND INPUT
     ROUTINE
170 NR =  VAL ( MID$ (DI$,DI,2)): IF
     NR < 1 THEN 160
180 RN = NR
190  ON RN GOSUB 10000,10050,10100
200  GOTO X: REM  X=START OF COMMAND
     INPUT ROUTINE
9997 :
9998  REM *** START OF ROOM DESCRIPTI
     ONS
9999 :
10000  PRINT "(ROOM DESCRIPTION)"
10010 DI$ = "00111400"
10020  RETURN
```

Program 6.2.

Line-by-line analysis

Although this second 'direct revaluation' program is longer than the routine in Program 6.1 it does save space in the long run because the actual movement codes - in DI$ - are shorter. Thus line 10010 of Program 6.1 takes up 22 bytes where line 10010 of Program 6.2 uses only 19 bytes (including the line header block in each case). As I said in the notes on Program 6.1, these routines won't do anything unless they are part of a complete Command Input/Revaluation/Room Description and Movement Code combination. So let's take a closer look at Program 6.2:

Lines 100-110: These lines are, as in Program 6.1, used to filter out commands longer than one letter plus commands I (Inventory) and L (Look).

Lines 120-150: Although we will end up by doing the same thing as we did in Program 6.1 - getting a positive or zero movement code - the system used here is rather different. Thus in these lines - assuming that we are not dealing with an illegal command - we do not get a new value straight away, but rather the location in DI$ where we can find the correct code. **Note:** In this example each

individual code has a length of two digits only, giving a maximum room code of 99. If you have more than 99 rooms on your map then the values for DI in these lines would be 1, 4, 7 and 10 and DI$ would be *twelve* characters long. Finally, in each line, when a value for DI had been collected execution moves on to line 170.

Line 160: As in Program 6.1 this message has been worded to deal with unsuccessful moves (which we will come to in a moment) and illegal commands. In either case program execution will go back to the start of the command input routine.

Lines 170-180: If the program has reached this point then we can look for an actual movement code in DI$. In order to do this we take the numerical value of 2 characters in DI$ starting at the DIth character from the *left* in DI$. Thus, if N$ equalled "E" then DI would equal 5 and we would translate the 5th and 6th characters of DI$ into a numerical value to be assigned to NR (New Room). Next we check that NR has a value greater than zero. If it is, then this value is copied to PL (the Player's current Location variable) in line 180. If the value of NR is 0 then the program goes back to line 160 and the move is rejected (the value of of PL is *not* altered). **Notes:** When we use the VAL command, 'leading zeros', as they are called, will be ignored. Thus if VAL (MID$ (DI$, DI,2)) actually equals '02' this will be interpreted by the computer as plain 2. Also don't forget that if you are using this routine with a map containing more than 99 rooms the expression in line 170 must be altered to NR = VAL (MID$ (DI$,DI,3)).

Lines 190-200: Exactly the same points apply to these lines as mentioned in the notes for the same two lines in Program 6.1.

Lines 10000-10020: These three lines are *basically* the same as the statements at the same line numbers in Program 6.1, but notice the difference in line 10010. Instead of assigning values to four variables we now have a single string called DI$. Thus we are still telling the computer that it cannot move North (N=0 in Program 6.1), that moving South takes the player to room 11 (E=11), and so on, but we do it by packing all the information into one variable value and let line 170 sort it out for us. Incidentally, if you were using this line in a game which included room numbers *above* 99 then this line would have to read 10010 DI$="000011014000", that is: 000 for North, 011 for South, 014 for East and 000 for West.

You won't be surprised to hear there is a better way of dealing with

the movement codes than any I've discussed so far. It does, however, involve playing about with the 'innards' of the computer so it will be of considerable value to start by discussing just *how* the C64 does what it does. Unless you already have a good understanding of things like *page zero* and the way in which BASIC programs are stored and executed I would advise against skipping this next section.

Bits and bytes and PEEKs and POKEs

It is quite beyond the scope of this book to explain machine code programming in any great detail. Nevertheless there are times when a basic knowledge of the 'low level' operations of your computer can allow you to do things which would not otherwise be possible. This is one such time.

The first thing I want to look at here is a part of RAM your computer makes continual use of – the zero page. The name zero page derives from the fact that all the addresses in the first 256 bytes of the computer's memory take up only two Hex digits. The point is that the highest number which can be held in one byte (that is, one address) is 255, or Hex FF (usually written as $FF). Since the computer usually expects a *four*-digit hex number – that is *two* bytes which it processes one at a time – this means that the second byte it looks at in a zero page address, known as the high byte or MSB (Most Significant Byte) will equal zero. Obviously the computer can process one byte faster than two, which means that using the zero page for much of its work allows faster overall processing time than if it used, say, the area of memory just below ROM, which consists entirely of four-digit hex addresses in most machines.

Now if the computer uses zero page whenever possible for its own operations, we would logically expect to find things like 'pointers' somewhere on this page. A pointer consists of two adjoining locations (usually on the zero page) which store a two-byte number – the address of something in memory which the computer needs to keep track of while it is RUNning a BASIC program. The list of addresses which require a pointer include the start and end of the program in the BASIC storage area, the start of the variable table and of the array and string storage areas, etc., etc. The two pointers we shall be using in the next section are (a) the START OF BASIC pointer and (b) the TOP OF PROGRAM pointer.

The second thing we need to know about what goes on inside the

computer concerns the actual storage of BASIC programs.

Have you ever wondered how a computer knows where to look for the line it is sent to by the GOTO and GOSUB commands? Or where it stores the number for each line of a BASIC program? The answer to these, and many other question, can be found in what I call the 'line header block'.

Believe it or not, every line of a BASIC program starts with a five-byte block that looks something like Fig. 6.1. The values are, of course, all in hexadecimal. The five bytes have the following functions:

```
00
19
08
10
00
```

Fig. 6.1. A BASIC program 'line header block' (decimal values).

Byte 1 – 00. This will be either the start of the program or the EOL, the End Of Line byte of the previous line of code. This byte serves two purposes. Firstly it tells the interpreter that the command or statement in the previous bytes is complete and should be processed. Secondly, the computer understands that it has reached the end of a line and that the next four bytes are for reference and *not* part of an instruction.

Byte 2 – 19. This is the first half of the address at which the header block for the *next* line of BASIC commences. This first byte is the lower half of the address (which is actually $0813), and is known as the 'low byte'.
(**Note:** It is worth remembering that computers habitually store two-byte numbers in this back-to-front fashion.)

Byte 3 – 08. The 'high byte' of the address of the next header block. This address is stored at the start of each line so that the computer can 'jump' from block to block when it is looking for a particular line during a GOTO or GOSUB, rather than having to scan through everything in the program.

Byte 4 – 10. Using the standard back-to-front format this is the low byte of the actual line number. In this example the line number is 10 ($0A in hexadecimal).

Byte 5 – 00. The last byte of the header block is the 'high byte' of the line number. It only comes into use for line numbers above 255 (decimal). The fact that the whole line number must be stored in two

bytes is the reason why BASIC programs cannot have lines numbered above 65535 (or Hex FFFF – the maximum value storable in two bytes).

With these two relatively simple pieces of information in mind we can start fooling the computer into doing what we want – without putting the program at risk!

Now you see it – now you don't

Learning to juggle with the innards of a computer is very much a matter of getting plenty of 'hands on' experience. So this section is almost entirely practical. That means that if you haven't got your computer switched on at the moment, then now's the time to do it.

Now, if everything is ready, let's begin. The first thing to do is to type in a one-line program (which does absolutely nothing):

```
10DATA12,40,50,100
```

Now enter the following line – without a line number – and press <<RETURN>>:

```
FORX=2048TO2069:?X" "PEEK(X),:NEXT
```

(Notice that in both cases there are no blank spaces in the line, except the one between the quotation marks.) All being well, you should now see the two columns on the left of Fig. 6.2 appear in a four-column display:

DECIMAL ADDRESS	DECIMAL CONTENTS	MEANING	HEXIDECIMAL ADDRESS
2048	0	0	$0800
2049	19	19	$0801
2050	8	8	$0802
2051	10	10	$0803
2052	0	0	$0804
2053	131	DATA	$0805
2054	49	ASC("1")	$0806
2055	50	ASC("2")	$0807
2056	44	ASC(",")	$0808
2057	52	ASC("4")	$0809
2058	48	ASC("0")	$080A
2059	44	ASC(",")	$080B
2060	53	ASC("5")	$080C

```
2061        48          ASC("0")    $080D
2062        44          ASC(",")    $080E
2063        49          ASC("1")    $080F
2064        48          ASC("0")    $0810
2065        48          ASC("0")    $0811
2066        0           End         $0812
2067        0           of          $8013  ($13=19 Dec.)
2068        0           Line        $0814
2069        88          bytes       $0815
```

Fig. 6.2. Memory contents for line 10.

No doubt you will recognise the first five items from the earlier example. They are, of course, the 'header block' for our line. And if you think about it for a moment it might seem reasonable to suppose that the contents of bytes 2049 and 2050 could be altered to point to any address that you liked. In other words, why not set them for some address a hundred or so bytes further on, and then store your own data in the 'hole' that you've created. Unfortunately

```
2048   0                2049   19
2050   8                2051   10
2052   0                2053   131
2053   49               2055   50
2056   44               2057   52
2058   48               2059   44
2060   53               2061   48
2062   44               2063   49
2064   48               2065   48
2066   0                2067   0
2068   0                2069   88
```

Fig. 6.3. Screen display of memory for line 10.

there are two problems associated with this. Firstly the process of SAVEing and re-LOADing will reset the header blocks to their original contents. Secondly, while the computer does read each header block pointer during a LISTing, when the program is RUN the computer's internal bloodhound – the text pointer that I referred to earlier – moves remorselessly from byte to byte. The only time that the text pointer bothers with the header block is when it is involved in temporarily storing it, or when it is looking for a prticular line in a GOTO or GOSUB. The next exercise will show you what I mean.

In the demonstration which follows we will insert a 'hole' into a three line program. Again the program must be entered *exactly* as given here – with no gaps. First type in

```
5REM0000,00
10A=10
```

Then re-enter the direct mode display line, only this time make it read

```
FORX=2048TO2091:?X" "PEEK(X),:NEXT
```

You should now see:

2048	0	2049	14
2050	8	2051	5
2052	0	2053	143
2054	48	2055	48
2056	48	2057	48
2058	44	2059	48
2060	48	2061	0
2062	23	2063	8
2064	10	2065	0
2066	65	2067	178
2068	49	2069	48
2070	0	2071	0
2072	0	2073	88
2074	0	2075	140
2076	1	2077	208
2078	0	2079	0
2080	43	2081	43
2082	43	2083	43
2084	43	2085	43
2086	43	2087	43
2088	43	2089	43
2090	43	2091	43

Fig. 6.4. Screen display of memory bytes 2048–2091.

The important locations to look at here are, firstly, bytes 2062 and 2063. They are the address bytes of the header block of line 10, pointing to the start of the next line of the program – $0823 (2071 decimal). Ignore the 8 for a moment and just add 23 to 2048 – though here this is the second of three End Of Program bytes because there is no more program. I'll deal with this in more detail in a moment. The second set of locations to note is 2073 to 2079. These will be holding different values in your display – probably all will contain 43 if you've just switched your computer on. Whatever values you do have in these locations note them down – you'll want to check them again in a moment.

Lastly, check locations 2080 to 2091. Unless you've overwritten

another program, as I had, this area will probably also contain the value 43 in every location.

Notice that the header block of line 10 – from 2061 to 2065 (inclusive) – is pointing at 2071 (ignore the 8 at 2063 and simply add 23 to the base address, 2048). Although our mini-program actually ends with the *three* End Of Program bytes in 2070 2071 and 2072, if we were to add another line in the normal way it would be tacked on starting at 2071, not at 2072 or 2073, because there can only be one zero between lines. Now we will add another line, but we'll start it at 2080, not at 2071. And for the sake of clarity we'll make it the same as line 10, apart from the line number. We can do this in two ways. For the moment let's stick to direct access and do it 'by hand', so to speak, by POKEing it in.

First of all we have to create a new header block (I'll explain why in a moment). So POKE the first five addresses like this:

```
POKE2080,0
POKE2081,42
POKE2082,8
POKE2083,20
POKE2084,0
```

and then copy the contents of line 10 plus the three EOP bytes:

```
POKE2085,65 - A
POKE2086,178 - =
POKE2087,49 - 1
POKE2088,48 - 0
POKE2089,0
POKE2090,0
POKE2091,0
```

If you now enter the line

```
FORX=2048TO2091:?X" "PEEK(X),:NEXT
```

you'll see that 2085-2091 is indeed a copy of bytes 2066-2072. And all we need to do to complete the job is to move the EOP pointer up to the new position, that is, one byte after the third zero. 2092 minus 2048 = 44, so POKE 45,44 and there we are. We've added a new line to our program – at the far side of a seven byte gap – and the computer is none the wiser. To prove the point just enter LIST and see what you get.

Whoops! What you didn't get was line 20. Have you been wasting your time? Am I out of my mind? Is the header block of line 10

pointing to the start of the new line? No, no and NO!! The low byte of the address at the top of line 10 (byte 2062) still contains 23. Enter ?PEEK(2062) and you'll see what I mean. To point it to the right location enter POKE 2062,33 and type LIST again. And there's line 20. The program is complete.

So far, so good. But we still have one more item to deal with. For if we were to SAVE our program and then re-LOAD it we would find that the byte of the line header block at 2061 had been re-set to 23. And that's why we need line 5.

Up until now line 5 has looked rather redundant. Its purpose, however, is to undo the housekeeping work that the computer does when it LOADs a modified program. At this point, then, (and still with no blank spaces) line 5 should be re-entered as:

```
5POKE2062,33
```

Obviously just having the line in the program won't do anything. But as long as the program is RUN after being LOADed byte 2061 will be re-set correctly (from our point of view) and subsequent attempts to LIST the program will produce the complete, modified listing.

If you've got it - flaunt it

This is all great fun, but how do we put it to practical use?

It is possible, as I said at the start of this chapter, to hide data either inside a program or at the end of it. And in both cases the data can be made virtually invisible. Let's start with the more practical option - storing data at the end of a program. In order to do this we need only take account of one set of pointers - at locations 45 and 46 - which hold the address of the point at which any BASIC program ends and its VLT, the *Variable List Table*, begins. (See Ian Sinclair's book *Introducing Commodore 64 Machine Code* for a detailed explanation of the VLT.)

The following program, though extremely simple to look at, gives an excellent demonstration of the way in which data may be POKEd to the end of a program without leaving any obvious trace.

Note: this program must be entered in three parts. It *must* also be entered exactly as shown - with *no* blank spaces in any line.)

Part 1:

```
5IFPEEK (45)>000THEN1010
10FORX=0000TO0000
60?CHR$(PEEK(X))"(2 spaces)";:NEXT
```

Well that looks pretty vague, doesn't it? The reason for all those zeros harks back to our last experiment – the line is made up to size with zeros because we don't yet know what numbers to use. But we can find one of the numbers that we'll need for the next line simply by checking the Top Of Program pointer. So enter

```
?PEEK(46)*256+PEEK(45)
```

and check the result. You should get the value 2108. But this will include the three zero bytes which mark the end of any program. In other words the true end of the program is at 2108 minus 4 (2104). What we actually want, however, is the location of the first byte of the address in the next header block. Given that the header block must start with a zero the location we're searching for must be halfway between these two addresses, at 2106.

To complete the first half of the program – those lines which will still exist after the program has been modified – we must add one more line which includes the address of its own header block:

```
70POKE2106,00:END
```

and then find out where this section of the program ends. So enter

```
PRINTPEEK(46)*256+PEEK(45),PEEK(45)
```

and you should get

```
2123        75
```

Now, the address section of this printout – 2123 – is, of course, the *fourth* byte after the last byte of actual program code. If we PEEKed at locations 2120 to 2122 they would all show up as zeros. So if we added another line, as we will in a moment, it would be tacked on at 2121, not at 2123. Thus the first byte of DATA should be POKEd at 2121. The first bytes of data, however, will be two zeros, to replace those that are overlaid when the next line is added (if we didn't do this the interpreter would think it had found the next line and try to LIST the DATA as a command!). The actual DATA that we wish to recover in lines 50-60 will indeed start at 2123. And that's the first of the two loop controllers to be entered in line 50. The second number, given that there are *ten* items of relevant DATA, can be found by adding 9 to this base address (base address plus nine others equals ten). Line 50 can now be altered, then, to read:

```
50FORX=2123TO2132
```

Next we must have a new NEXT LINE address for the header block

of the last line of the permanent program (line 70). In order to get this we have to take the address of the last piece of relevant data – 2132 – and add 2. In this way when the computer jumps over the POKEd data it will land on the middle byte of three zeros, just as it would if it came at the real end of a program. In other words the computer must be made to go 11 bytes beyond the original End Of Program position, which showed as 75, and the second byte of line 70's header block must read 86. Lastly, the permanent program will appear to the computer to end two bytes beyond that last address, at 2048+88, and 88 is the number to check for in the Low Byte of the Top Of Program pointer at location 45.

The final version of the permanent program should now look like this:

```
5IFPEEK(45)>(one space)88THEN1010
50FORX=2123TO2132
60PRINTCHR$(PEEK(X))"(two spaces)";:NEXT
70POKE2106,87:END
```

At this point the rest of the program can be added in the normal manner as no DATA has yet been transferred and no 'hole' exists. Part 2 of the program looks like this:

```
80 REM
1000 DATA 0,0,65,66,67,68,69,70,71,72,73,74,
     0,0, 0,88
1010 BA = 2120
1020 FOR X = 1 TO 16
1030 READ NO: POKE BA + X,NO
1040 NEXT
1050 POKE 45,88: GOTO 50
```

The lines we have added are all straightforward except, perhaps, for line 1050. Here we are moving the Top Of Program pointer *down* to the address of the last byte of POKEd data – the 88 – so we can SAVE the program. If this routine came close to a 'page boundary', where the High Byte of the Top Of Program pointer might also be affected, then the value in location 46 might also need to be altered by subtracting 1 from its current value.

Please note that once this program has been RUN none of the lines below 80 may be altered, unless you wish to change the IF in line 5 to REM (they both take up just one byte) just to confuse things for anyone reading the final program. If you make any other alteration to the actual number of *bytes* in lines 5-70 inclusive you will need to redefine the values in lines 5, 50 and 70.

The program with a hole in it

I said earlier that you could POKE data into the middle of a program and still have it look right. I'm afraid I may have overstated my case.

It is certainly true that on *some* computers you could make a 'data hole' in any part of your program and it wouldn't make a scrap of difference. Indeed, my earlier example may have seemed to show that you could do such things on the C64. In fact you can't . For while such programs will LIST quite normally, nothing beyond the hole will actually RUN without a considerable amount of tampering with various locations.

The problem here is that the C64 doesn't use just one set of *text pointers*. If it did, then we could use a positively tiny machine code program – about ten bytes long – whenever we wanted to jump from one side of the hole to the other. In the C64, however, lines of BASIC are not read by the interpreter in the Basic Program area. Instead they are moved to a rather small buffer at the bottom of 'page' 3 (locations 512-600) and TEXTPOINTER points to this buffer.

The final trick that I offer in this chapter, then, is a way of entering fake lines beyond a hole without having to do it on a byte-by-byte basis. Like all the best tricks it is actually extremely simple, though you may need a bit of practice to get it right first time, every time. All that you need to do is to temporarily move the START OF BASIC up to the end of your POKEd data. To be precise, if we were using it with the last program then we would alter location 43 (the Low Byte of the START OF BASIC pointer) to the same value held in the modified version of location 2106 – 84 – and place that address plus 2 in location 45. In this way the computer would think it was starting a new program and would itself give each additional line of code a 'correct' header block so that the added lines would LIST correctly. Once you have finished adding lines, simply enter POKE 43,1:POKE44,8 and the START OF BASIC pointer will again point to the correct start of your program. The values in the TOP OF BASIC pointer should be left exactly as they are.

Oh, and don't forget, you'll still need to include a statement *before* the hole which will make the necessary adjustment to the pre-hole line header block; this will take effect whenever the program is RUN (see line 70 above). Which reminds me of another interesting point before we close. What happens if someone tries to LIST the program before RUNning it – won't they find the POKEd data? Actually no, because the computer will try to interpret the data as lines of BASIC and go absolutely haywire!

Chapter Seven
A Code in Time Saves...

Leaving aside the most basic details of layout and programming, the most important part of an adventure is the composition of the room descriptions. The originality, humour and inventiveness that goes into the writing of these descriptions can, *sometimes* save even a third-rate game from being a total disaster. But there is a limitation involved in this part of the adventure writing process – lack of RAM space.

As I mentioned in the first chapter, one of the major factors involved in moving adventure games from mainframe computers over to micros was the gross inequality of RAM space available on the two types of machine. It's quite true, of course, that even mainframe computers (despite their terrific size) once had their RAM capacity reckoned in tens of kilobytes rather than hundreds and thousands. The early machines took up vast areas of floor space because they were made up of wires and valves rather than the integrated circuits of today. But by the time the first adventure games were being developed – many of them on machines like the DEC PDP/11 – transistors had taken over from valves, and had in turn been overtaken by the silicon chip. Memory space in the average machine was increasing by leaps and bounds.

But why, you may ask, was so much RAM space needed? After all, the main control programs were much the same size as they are today, though often written in languages like FORTRAN rather than in BASIC or machine code. To find the answer to this question we need only look around and see the difference between a good adventure and one of the 'also rans'. Take this passage from a recent review:

> '... the room descriptions are *far too short*, giving only the name of each location, a brief description and a list of possible moves.'

And that's where the memory goes in the best games – on the room descriptions.

Don't get me wrong. I'm not saying that *lengthy* descriptions are necessarily *good* descriptions. But even where all the other factors in a game are as well thought-out and presented as they can be, where text is kept as brief as possible while being as effective as possible, thousands of bytes are needed to produce the kind of text screens that will make the player's situation seem both realistic and believable.

If you've ever had the chance to play a top-rated game, or even if you've only seen sample displays in magazine articles and reviews, you will know that each full-length room description takes up most of the screen. That means that each *long* description (as opposed to the short descriptions used by some programs when you re-enter a room) is something like 300-400 characters long.

'Hang on there,' gasps the voice. 'Two hundred room descriptions at 400 bytes a time. That's about 80K!' Where do you put all those bytes in a 32K-48K computer?'

If we stick to what might be called the 'conventional' approach then there are three possible alternatives, depending on which machine (and which peripherals) you are using:

(1) If you have at least one disk drive then your problems are more or less over. Room descriptions may be stored as separate records in a random access file. In this case the size of each room description, and the total number of descriptions used in the course of a game, will depend on nothing more than how much information you can get on each disk, how many disks you use and the size of the game that the control program can handle.

(2) The second alternative is to store the room descriptions in the control program itself (in the form of DATA statements). The problem comes when the data is transferred to an array, as it must be each time the game is RUN. Suddenly you begin to lose RAM space at an alarming rate, for the entire list of descriptions is now in two separate blocks of RAM – in the DATA statements *and* in the array. Quite amazingly, several games released as late as 1983 still used that archaic form of storage!

(3) If your computer can handle such tasks, the room descriptions can be set up in a separate program which holds nothing but room descriptions (as DATA statements), which are transferred to an array. Once the transfer is complete it is possible to SAVE the array

alone, and the 'set up' program is then discarded. Thereafter the control program can LOAD the array straight into the memory and does not need to carry the identical set of DATA statements.

As we will see in a moment, this third alternative, subjected to a little extra manipulation, can be a very effective way of storing room descriptions – or any other text for that matter – even in comparison with a disk drive.

Time and space

Before we get down to discussing the second of our 'super space-saving storage systems' let's take a quick look at the advantages and disadvantages of the various methods of storage.

If, for example, we use a single disk drive, we can count on something in excess of 80K of storage space on each disk. This just happens to be the exact size of the storage space needed for a game with around 200 room descriptions of a satisfactory length.

In practice, of course, the amount of storage space per disk side has been rising steadily over the last few years. Out of more than sixty drives included in a December 1983 trade list only six offered less than 105K storage per disk side. And only one offered less than 90K. In fact the Commodore 1541 drives – offering around 170K of storage – are now on the low side of average, with several ordinary 5-inch floppy drives offering as much as 500K or more for the same kind of price. With this amount of storage available it is possible to store not only the entire set of room descriptions, plus the movement codes, on one disk, but also the main game program. In this way the computer can constantly access the drive to collect each room description and the set of movement codes for each new move as and when they are needed. This means that while the game is RUNning we need only enough space in RAM, to hold one room description and one set of movement codes, leaving almost all the programming area free for the control program.

This sounds pretty good, but there is a price to pay – a wait of several seconds on each move while the computer (a) turns the disk drive on, (b) collects the room description from one or two records, (c) collects the relevant set of movement codes, and (d) waits for the drive to return control. And on top of all this you must remember that the drive has to *find* each record before it can READ it – that's a lot of wear on the BAM sectors and the 'read head'.

(**Note:** because cassettes can only read information *sequentially* – they must read each file from the start in order to find a particular record – the amount of time needed to collect a given room description from tape would be measured in minutes rather than in seconds. In other words, it would be totally impractical.)

So, the first method is at least practical. Method (2) – transferring DATA to an array – is so costly in terms of RAM space that it isn't worth serious consideration. At least, it shouldn't be! Which brings us to the third alternative.

Method (3) – the SAVEd array method – has as its main advantages over disk access (a) the almost instantaneous display of each room description as it is called up, and (b) an enormous saving of wear and tear on the storage facilities. Unfortunately it also has two major limitations. Firstly it requires that *all* the room descriptions be in memory at all times. Secondly, it isn't a trick that you can use very easily on the C64. So, we come back to our original question: how do we get all that text into a limited amount of RAM space?

Of course, we could simply 'slim down' the text, but to do this would require a pretty drastic bit of editing. Lose seventy-five per cent of the text space mentioned above and you still have 20K of text that has to be stored somewhere. And maybe you've lost a good deal of the 'punch' of the original text in the meantime!

For reasons that totally escape me for the moment, it has taken something like three or four years – since the first micro-adventures arrived on the market – for someone to come up with what is really the 'obvious' solution to this problem.

I know I've used this 'it's so obvious you'll kick yourself' line before, but this time it really is *obvious*. Because the method involved, first successfully used by Level 9 in their range of adventure games, is nothing more or less than an adaptation of the routine that every micro already contains – a coding routine based on 'tokenised' keywords.

Packing 'em in

Before I get down to the details of this routine I'd like to give you some idea how effective it is. In order to do this I would ask you to look at the amount of text contained in this chapter, from the first word of the title to the last word in this sentence.

Now, if we include everything that appears in that block of text,

which includes letters, numbers, punctuation marks (and even the blank spaces between words), then we have a total of just over 9,000 characters. So let's suppose we can only encode, or tokenise, one set of the letters - t, h and e - so that only one byte will be needed to signify each occurrence of 'the'. What kind of result will we get?

Actually this particular letter group is quite common in the English language - *the, the*n, ra*the*r, o*the*r, etc., etc. - so it's not much of a surprise to find it occurs about 146 times (including a few The's) in this passage. In other words, out of 9,000 characters a total of 438 are used for the sequence *the*. Not many? Actually they amount to a little under five per cent of the total. If we can tokenise this one group, then, we will have saved two-thirds of the space it takes up.

In practical terms, tokenising the letter group *the* in this passage would save 3.2 per cent of the space. And that's not all. A closer study of the text shows that, in this passage at least, the letters *the* usually occur at the start of a word (or on their own). I haven't done an exact count, but let's suppose that on 125 occasions the actual character group consists of 'the'. In this case we can tokenise one hundred and twenty-five *four*-letter groups, a total saving of 4.1 per cent of the text. If we apply that to the 80K description file mentioned earlier we're now talking about saving something like 3.2K with just *one* code!

Perhaps 3.2K still doesn't look like a very big saving on 80K, but there are two factors which need to be taken into consideration. Firstly we're not restricted to using just three or four codes. We can, in fact, go as high as 127. This doesn't mean you can tokenise everything in sight and end up with just a couple of hundred tokens in your room description array. But it does suggest we can knock a pretty big hole in that 80K. Secondly we are not restricted to three and four-letter groupings. The programs we will be coming to in a moment can handle letter groups of any reasonable size. Thus, if we were using a 'one-for-three' system we could not hope to reduce our text by more than 66.6 per cent at the absolute maximum. Using a 'one-for-four' system the ideal saving rises to 75 per cent. At 'one-for-five' it rises to 80 per cent, and so on. And for added efficiency we can actually mix up codes of differing lengths in the same program. The only common factor will be that each letter grouping, regardless of size, will be *stored* as just one byte!

Program 7.1: Sardines - the computer version

So how do we go about this 'text crunching'? First, we need to study the ASCII codes for the C64. These are the *numerical* values that the computer uses to store letters, symbols and numbers where they appear in a program listing or as text. Normally all these codes come *below* 127, so we can use values above 127 for our codes without fouling up the computer's normal operations. What we are going to do involves three operations:

(1) Search every room description to discover which letter groups, if encoded, will give us the greatest savings
(2) Set up a code array so that each room description can be tokenised
(3) Introduce a decode routine so that our program can translate any decoded passage back into its original form for display.

This will involve three separate programs, the first two of which are, to put it mildly, an absolute pain! The programs themselves are fairly simple and straightforward; it's putting them to use which takes so much time. Nevertheless, the results will, I promise you, be well worth the effort. And just to prove the point I want to start off with a very short introductory program which shows the 'encode/decode' process at work. (By the way - I've deliberately misspelt my name here to give the program a little extra work to do.)

```
]LIST

1  REM *****  ENCODE/DECODE DEMO  ***
     **
2  :
3  :
8  REM *** ENCODER
9  :
10 A$ = "ANDRAW BRADBURY"
20 A$(1) = "RA"
30  FOR X = 1 TO 15
40  IF  MID$ (A$,X,2) = A$(1) THEN B$
      = B$ +  CHR$ (128):X = X + 1: GOTO
     60
50 B$ = B$ +  MID$ (A$,X,1)
60  PRINT X"  "B$
70  NEXT
97 :
```

```
98  REM *** DECODER
99  :
100   FOR Y = 1 TO  LEN (B$)
110 K =  ASC ( MID$ (B$,Y,1))
120   IF K > 127 THEN  PRINT A$(K - 12
      7);: GOTO 140
130   PRINT  MID$ (B$,Y,1);
140   NEXT
```

Program 7.1.

Line-by-line analysis

Line 10: In the two later programs this string would be the one you INPUT from the keyboard. In the decode program it will be one of the room descriptions in the memory.

Line 20: This arrayed string represents one of the code strings to be found by the String Analyser (Program 7.2) and subsequently stored as CD$(X)

Lines 30-70: A simple search loop which 'steps' through the string looking for a match to A$(1). It moves one character at a time, but always reads two characters at a time (X and X+1). If a match is found, a code character is added to B$ and the search jumps forward *two* letters - one letter in line 40 (X=X+1) and a second when it hits the NEXT command in line 70. If no match is found the current letter is added to B$ - line 50 - and the loop continues. Line 60 has been inserted simply to show you what is happening during the loop.

Lines 100-140: When you've gone through the String Analyser and Encoder programs you may wonder how long the *decoder* program is going to be. Well, here's your answer - this is all it takes. These lines repeat the search suggested by lines 40-50, only now we are only looking for the 'odd men out' - ASCII codes higher than 127. When found, the coded byte is replaced by the appropriate letter group and the loop continues until finished.

Eagle-eyed readers may notice that the last line of the display - the decoded version of B$ - prints out slightly slower than normal. This is an unavoidable delay caused by line 110 and the IF check in line 120 which must occur before each letter (or letter group) is sent to the screen.

Program 7.2 Checking it out

Let me say right at the start that this program takes a *long* time to RUN. The actual time will depend upon how many strings you analyse at any one time: the 'drag factor' is the constant searching of the SA$() array for matching letter groups.

```
1  REM *****  $TRING ANALYSER  *****
2 :
3 :
8  REM *** PREPARE STORAGE ARRAYS
9 :
10  DIM SA$(2000),SI(2000):CA = 1
97 :
98  REM *** INPUT STRINGS
99 :
100  FOR Y = 1 TO 255
120  PRINT "[CLEAR][DOWN * 2]PLEASE E
     NTER STRING NUMBER "Y" NOW: [DOW
     N]"
130 IN$ = ""
140  FOR X = 1 TO 255
150  GET I$: IF I$ = "" THEN 150
160  IF I$ =  CHR$ (13) THEN IN$ =  LEFT$
     (IN$,X   1):X = 255: GOTO 190
170  IF I$ =  CHR$ (20) THEN  PRINT I
     $:IN$ =  LEFT$ (IN$,X - 2):X = X
      - 2: GOTO 190
180 IN$ = IN$ + I$: PRINT I$;
190  NEXT
196 :
197  REM *** DISPLAY IN$ FOR
198  REM      CORRECTIONS
199 :
200 Z$ = "": PRINT "[CLEAR][DOWN * 2]
     "IN$"[DOWN]"
210  PRINT "IS THIS CORRECT (Y OR N)?
      ";
220  GET Z$: IF Z$ = "" THEN 220
230  PRINT Z$: IF Z$ <  > "Y" THEN 12
     0
240  PRINT "[DOWN]THANK YOU."
297 :
298  REM *** AND BREAK IT DOWN
299 :
300  FOR Q = 1 TO  LEN (IN$) - 2
```

```
310 HO$ =  MID$ (IN$,Q,3)
320  FOR HA = 1 TO CA
330  IF HO$ = SA$(HA) THEN SI(HA) = S
     I(HA) + 1:HA = CA: GOTO 360
340  NEXT HA
350 SA$(CA) = HO$:CA = CA + 1: GOTO 3
     70
360  NEXT HA
370  NEXT Q
397 :
398  REM *** GET ANOTHER STRING?
399 :
400  PRINT "[DOWN]ANY MORE $TRINGS TO
      ANALYSE (Y OR N)? ";
410  GET Z$: IF Z$ = "Y" THEN  PRINT
     " ": NEXT Y
420 Y = 255: NEXT Y
497 :
498  REM *** OR DISPLAY RESULTS
499 :
500  FOR Y = 1 TO CA STEP 15
510  PRINT "[CLEAR]"
520  FOR X = Y TO Y + 14
530  PRINT SA$(X),SI(X)
540  NEXT
550  PRINT "[DOWN * 2][RVS]PRESS ANY
     KEY TO CONTINUE[OFF]";
560  GET Z$: IF Z$ = "" THEN 560
570  PRINT " ": NEXT Y
```

Program 7.2.

By the way, the number of occurrences of each letter group will always be the correct total *minus* 1 since the start value of any element of SI() will always be 0 and they are not incremented until a repeat group is found. If you prefer total accuracy then alter line 530 to

```
530 PRINT SA$(X),SI(X)+1
```

Line-by-line analysis

Line 10: If you're prepared for a long session then the size of these arrays can be increased considerably. The maximum size will depend on the length of the letter groups you want to search for. At the same time, however, the time it takes to scan the SA$() array for a match can be very long indeed when you have a thousand or more elements filled.

Lines 100-190: The program is currently set to handle a maximum of 255 room descriptions, though this limit could be raised if you're truly ambitious. The size of the X loop should *not* be increased, however, as doing so could crash the program.

Lines 160 and 170 are designed to deal with the end of string RETURN and any deletions you may wish to make. Line 160 simply clears the loop and moves on to the next stage. Line 170 deals with deletions by knocking off the last *two* letters of the string, setting the value of X to X-2 and then re-entering the loop. Taking out two letters rather than one *is* necessary - try taking out just one and you'll see what I mean.

Lines 200-240: Present the completed (?) string for approval. Line 240 is there just to reassure you that the program is still in operation when the waits get longer.

Lines 300-370: As in the demonstration program we now have to step through the input string looking for a match to an element of SA$(). In this case we are looking for three-letter groups so, in line 300, we must not go beyond the third from last letter or the routine crashes. The size of groups you're looking for is controlled by the last number in the brackets in line 310 and this *must* be 1 greater than the number at the end of line 300.

The variable CA indicates the lowest *empty* element of SA$(). In lines 330-340 SA$()'s elements are searched for a match to HO$. If a match is found then the counter for that element - SI(HA) - is incremented, HA is set to its highest possible value (CA) and the program jumps to complete the loop at 360 so that the Q loop can go again. If no match is found then the first empty element of SA$() is filled, CA is incremented and the Q loop is repeated if necessary.

Lines 400-420: Ask if there are any more strings to be analysed. If 'Yes' then the Y loop goes again. If not then the Y loop is completed and ...

Lines 500-570: The results of the string analysis are displayed, 15 elements at a time. (I'm afraid you'll have to note down which are the commonest letter groupings by hand.)

Program 7.3: CRRUNCCCH!

Having analysed the contents of your room descriptions you might well take time out to see if you can't re-word a few passages to fit in

with your top 127 letter groups. Also, if you've run more than one analysis - to test for letter groups of different lengths - you may want to check that none of the groups overlap. If they do you'll obviously go for the one which offers the greater saving, but you'll have to discard the other.

The next job is to actually start encoding some strings. Which brings us to the ENCODER. By the way, the whole of the first part of this program, with the exception of line 10, is exactly the same as in the String Analyser, so you can save a lot of time by SAVEing the last program and then chopping off lines 300 onwards to prepare for this next program.

```
]LIST

1  REM *****  $TRING ENCODER  *****
2 :
3 :
8  REM *** PREPARE STORAGE ARRAYS
9 :
10  DIM RD$(255),CD$(127):LC = 127:CL
      = 3
17 :
18  REM *** AND FILL CODE ARRAY
19 :
20  FOR X = 1 TO LC
30  READ CD$(X)
40  NEXT
97 :
98  REM *** INPUT STRINGS
99 :
100  FOR Y = 1 TO 255
120  PRINT "[CLEAR][DOWN * 2]PLEASE E
     NTER STRING NUMBER "Y" NOW: [DOW
     N]"
130 IN$ = ""
140  FOR X = 1 TO 255
150  GET I$: IF I$ = "" THEN 150
160  IF I$ =  CHR$ (13) THEN IN$ =  LEFT$
     (IN$,X   1):X = 255: GOTO 190
170  IF I$ =  CHR$ (20) THEN  PRINT I
     $:IN$ =  LEFT$ (IN$,X - 2):X = X
      - 2: GOTO 190
180 IN$ = IN$ + I$: PRINT I$;
190  NEXT
196 :
```

```
197  REM *** DISPLAY IN$ FOR
198  REM      CORRECTIONS
199 :
200 Z$ = "": PRINT "[CLEAR][DOWN * 2]
    "IN$"[DOWN]"
210  PRINT "IS THIS CORRECT (Y OR N)?
     ";
220  GET Z$: IF Z$ = "" THEN 220
230  PRINT Z$: IF Z$ <  > "Y" THEN 12
    0
240  PRINT "[DOWN]THANK YOU."
297 :
298  REM *** AND INSERT CODES
299 :
300 HO$ = ""
310  FOR Q = 1 TO  LEN (IN$)   CL
320  FOR HA = 1 TO LC
330  IF  MID$ (IN$,Q,CL) = CD$(HA) THEN
    HO$ = HO$ +  CHR$ (HA + 127):HA =
    LC: NEXT HA:Q = Q + (CL - 1): GOTO
    360
340  NEXT HA
350 HO$ = HO$ +  MID$ (IN$,Q,1)
360  NEXT Q
396 :
397  REM *** STORE ENCODED $TRING
398  REM      AND GO ROUND AGAIN?
399 :
400 RD$(Y) = HO$
410  PRINT "[DOWN]ENCODE OLD$/NEW STR
    ING OR QUIT (O, N OR Q)? ";
420  GET Z$: IF Z$ = "" THEN 420
430  IF Z$ = "Q" THEN Y = 255: NEXT :
     END
440  IF Z$ = "N" THEN  PRINT " ": NEXT
    Y
450  IF Z$ <  > "O" THEN 410
460 IN$ = RD$(Y): INPUT "[DOWN]NEW CO
    DE LENGTH? ";CL
470  GOTO 300
1000  DATA  (CODEABLE LETTER GROUPS)
```

Program 7.3.

Line-by-line analysis

Line 10: The RD$() array is for the storage of encoded strings. CD$() should be DIMmed according to the number of coded groups

you are using. CL is the number of letters in coded strings in CD$(). For LC see below.

Lines 20-40: These lines are an addition to, but don't clash with anything in, the last program. This is, of course, a simple loop to load the CD$() array with the actual letter groups that are to be encoded. LC stands for Last Code; its highest *safe* value is 127 and should be set in line 10.

Lines 100-240: Exactly as in the String Analyser program.

Lines 300-360: As they stand, these lines are designed to use a version of CD$() in which all the codes are the same length. The routine at 400, however, allows multiple encoding of the same string where various elements of CD$() are of different lengths.

Line 330 does the actual encoding when a match for any part of CD$() is found in IN$. Note the statement Q=Q+(CL−1) at the end of this line. This value *must* be CL−1 - to jump a given number of (encoded) letters - as the number will be increased by 1 again at line 360. If no match is found then the letter at MID$(IN$,Q,1) is added to the encoder string as is.

Line 400: Stores the encoded string as RD$(Y)

Lines 410-470: You now have three options: to continue encoding the last-used string (RD$(Y)) using a new code length, to start over with a new string, or to QUIT. Line 430 handles QUIT (notice we still clear the Y loop even though the program is over). Line 440 re-activates the Y loop to go back for a new string. And lastly lines 460-470 re-assign RD$(Y) to IN$ and collect a new value for CL (Code Length) in order to return to the encoding routine.

'Hang on a minute. Now we've got it - what do we do with it?'

I thought you'd ask that. The answer is not exactly a piece of cake, but before I give it to you perhaps you'd like to try writing the routine yourself. Here are the main points that the necessary code must deal with:

(1) Reset the TOP OF MEMORY pointers to allow space for the stored strings in a 'safe' area.
(2) Working your way *backwards* through the encoded string - using a loop (what else!) - store the *last* string at the top of memory. In other words work your way down from 40959.
(3) Once the string is stored take a note of (a) its length, and (b) the address of the first byte, in Low byte/High byte form.
(4) If you've any strings left to store then go back to step (1).

(5) When all the strings have been stored enter a 'table' of sets of three-byte pointers below them so that working your way *up* in memory you have byte 1 (length of RD$(X)), byte 2 (High byte of starting address of RD$(X)), and byte 3 (Low byte of starting address).
(6) Make a note of the *start* address of this table – its length will be (number of rooms) * 3 – then raise the BOTTOM OF BASIC pointer to the byte immediately before the start of the table (re-set locations 43 and 44) and raise the TOP OF BASIC pointer to the end of the descriptions (re-set locations 45 and 46). Then SAVE everything in the normal way.
(7) Lastly, when the rest of your programming is completed, clear the BASIC Programming area and LOAD the game program. Then raise the BOTTOM OF BASIC pointer to the address used when you SAVEd the RD$() array and table, set the TOP OF BASIC pointer to the address two bytes above it and re-LOAD the table and arrays. Finally, lower the BOTTOM OF BASIC pointer to its normal position and SAVE the whole of the BASIC Program area.

To use this method your *game* program must contain the following features.
(1) As soon as the program starts RUNning it *must* set the TOP OF MEMORY pointer to the byte below the array table (alter locations 55 and 56) or everything at the top of the programming area will be overwritten. (Note: When re-setting the TOP OF MEMORY pointers always follow the operation with the CLR instruction to ensure that the computer recognises the new TOM location.)
(2) In order to reclaim any string you will need a printout loop which is set to read LEN(RD$(X)) letters starting at the address given by High byte *16+Low byte. You will find the necessary information by calculating the address of the length byte using the formula LEN BYTE = PEEK ((address of lowest byte in the table −3) + (room number * 3)). The address for the loop to start reading from will be at PEEK((LEN BYTE + 1) * 256) + PEEK(LEN BYTE +2). You should PEEK and PRINT from that address to that address + (value in LEN BYTE −1) because the first letter will be at byte 0, not at byte 1. In other words, your PEEK and PRINT loop should start with a line like

```
10 FOR X = (B * 256 +C) TO (B * 256 + C) + A
```

where A equals the value in the LEN byte, B is the High byte of the start-of-string address and C is the Low byte of the start-of-string address. I know this is rather complicated, but I hope I've explained

enough in the previous chapters for you to handle it without too much difficulty. And now, whether you've found the solution to our little puzzle or not, here is my version of the necessary code.

First, the whole program must begin by moving the TOP OF MEMORY down far enough to allow room for the strings that will be stored. This must be done in advance or we will certainly end up by overwriting both the latest INPUT string *and* the array which holds the code letter groups (CD$()). Obviously we can't know in advance just how large a space we will need, so let's err on the side of safety and give ourselves a full twenty thousand bytes (this can always be altered later if necessary – see below). So, we start by adding line 5 (or any line number *below* 10 – that is, before the arrays are set up):

```
5 POKE 56,81: POKE 55,223: CLR: TS = 40959
```

The top of BASIC memory is now situated at 20959 (PEEK(56) * 256 + PEEK(55)). The CLR command ensures that the computer recognises this alteration. And TS, used in the subroutine below, is initialised *after* the Variable, Array and String pointers have been cleared.

Next, line 440 of the encoder program needs to be changed. If you are moving on to encode a new string then this is the time to store 'Old String'. Line 440 should now look like this

```
440 IF Z$ = "N" THEN  GOSUB 2000: PRINT " ": NEXT
```

In fact we also have to change line 430 – for extra user-friendliness – but let's add the subroutine first, before we forget it:

```
2000 LE = LEN (RD$(Y))
2010 FOR X = 1 TO LE
2020 LL     =     TS     -     LE     +     X:     POKE
     LL,ASC(MID$(RD$(Y),X,1))
2030 NEXT
2040 TS = TS - LE
2050 PRINT LE, INT(TS / 256), TS - (INT((TS
     + 1) / 256) * 256)
2060 IF   (INT((TS + 1) / 256)) <= PEEK (56)
     AND   T - (INT((TS + 1) / 256) * 256) +
     1 < PEEK (55) THEN END
2070 RETURN
```

In line 2000 we get the length of the encoded string, but as a control figure for the loop and for our own information.

Lines 2010-2030 POKE the ASCII codes of each string into memory in *ascending* order ending at the current address whose value is given by TS.

Line 2040 re-sets the value of TS ready for the next storage operation. Note that the first byte of each string is not POKEd to - TS − LE but to TS − LE + 1, one address *higher* in memory. So address TS − LE is indeed the correct ending point for the next string down.

You'll remember that we are going to have to construct a reference table at the bottom of the room descriptions in order to find them again. Line 2050 prints out those values in the order that they will appear in the table. LE is the length of the string. INT((TS+1)/256) gives us the High byte value for the start address. TS−(INT((TS+1)/256)*256)+1 gives us the Low byte of the starting address, that is, the current value of TS *plus* 1.

Line 2060 is *very* important. If TS falls lower than the address whose values you POKEd into 56 and 55 at the start of the program – in this case 20959 – then you've probably just overwritten the top of the String storage area. I've chosen to let this occur – rather than waste the INPUT – but your POKEd strings have now left the 'safe' area and the program will therefore cease to operate to avoid overwriting the last set of POKEs. As long as you are still within the safe area, however, the program will RETURN to line 440.

And that leaves us with line 430. I've left this till last as its amended version will be much clearer in the light of the operations we've just dealt with. The first thing to note is the fact that, if we simply QUIT the program on this line, the last string to be encoded would not be stored. We must therefore include a GOSUB to the routine described above. If we are ending the encoding routine, moreover, we will also need to make up the table I've mentioned. This will need another subroutine of course. So, first we change line 430:

```
430 IF Z$ = "Q" THEN GOSUB 2000: GOSUB 3000: Y = 255:
NEXT: END
```

and then we add the 'Table Maker' subroutine follows:

```
3000 PRINT: INPUT"ENTER TABLE OR QUIT (T OR
     Q)? ";
3010 GET Z$: IF Z$ = "" THEN 3010
3020 IF Z$ < > "T" THEN RETURN
```

```
3030 TL = TS - (Y * 3) + 1: K = 1
3040 FOR X = TL TO TS STEP 3
3050 PRINT:  PRINT  K"  ";:  INPUT"STRING
     LENGTH? ";SL: POKE X,SL
3060 PRINT: PRINT K" ";: INPUT"ADDRESS HIGH
     BYTE? ";HB: POKE X + 1,HB
3070 PRINT:  PRINT  K" ";:INPUT"ADDRESS LOW
     BYTE? ";LB: POKE X + 2,LB
3080 K = K + 1
3090 NEXT
3100 PRINT:  PRINT"TABLE  COMPLETE.    BYTE
     BEFORE TABLE ="
3110 TL  = TL - 1: LT = TL - (INT(TL / 256)
     * 256): PRINT"LOW BYTE: ";LT
3120 PRINT"HIGH BYTE: ";INT(TL / 256)
3130 PRINT"START  OF  TABLE DECIMAL VALUE =
     ";TL + 1
3140 RETURN
```

Programs 7.4 and 7.5: Bringing it all back home

And that's just about it. Our last two programs, positive midgets compared the last String Analyser and the Encoder, are different versions of the kind of decoding routine to be included in the actual adventure game. Program 7.4 deals with encoded text held in an array. Program 7.5 handles room descriptions that have been POKEd into memory. They are both short, efficient, and almost exactly the same as the second part of the demonstration routine. Neither program will run quite as fast as a normal print statement, and Program 7.4 won't run quite as fast as the demo program, its actual speed depending on the size of the RN$() array. Nor, on the other hand, will they run anywhere near as slowly as the analyse and encoder programs, since we are able to look quite directly at both the contents of RN$(PL) and CD$(), rather than searching through the arrays, as we have a specific reference to the location of the relevant information in both.

```
]LIST

1  REM *****  DECODER #1  *****
2 :
3 :
4  REM *** RN$() - ROOM DESCRIPTIONS
5  REM      PL - PLAYER'S LOCATION
```

```
6  REM      CD$() - CODE LETTER GROUPS

7  REM      THIS SUBROUTINE IS USED
8  REM      WITH EACH SUCCESSFUL MOVE
9 :
100  FOR Y = 1 TO  LEN (RN$(PL))
110 K =  ASC ( MID$ (RN$(PL),Y,1)
120  IF K > 127 THEN  PRINT CD$(K - 1
     27);: GOTO 140
130  PRINT  MID$ (RN$(PL),Y,1);
140  NEXT
150  RETURN
```

Program 7.4.

Line-by-line analysis

Line 100: It is assumed that the room descriptions will finally be stored as elements of RN$(). It is RN$(PL), where PL stands for the player's current location, that is to be printed on the screen, one way or another.

Lines 110-130: Using K as a numerical variable for the temporary storage of ASCII codes we 'look' at and display, one at a time, the characters in RN$(PL). Unless, of course, the value passed to K is greater than 127. In that case we deduct 127 from the value of K and use the result as an index to the CD$() array to see which letter group should be printed.

Lines 140-150: Line 140 keeps the loop going until it is complete; when the whole of RN$(PL) has been printed. We then close off the string with a blank space (since it was left open in line 130) and then RETURN from what should indeed be a subroutine since it will be used so frequently.

Program 7.5

```
]LIST

1  REM *****  DECODER #2  *****
2 :
3 :
4  REM *** BA - TABLE START MINUS 3
5  REM      PL - PLAYER'S LOCATION
6  REM      CD$() - CODE LETTER GROUPS
```

```
7  REM      THIS SUBROUTINE IS USED
8  REM      WITH EACH SUCCESSFUL MOVE
9 :
16 :
17  REM *** INITIALISE BA AT START
18  REM             OF GAME
19 :
20 BA = 20959 - 3
97 :
98  REM *** READ 3 TABLE BYTES
99 :
100 TL = PL * 3 + BA
110 LE =  PEEK (TL):ST =  PEEK (TL +
     1) * 256 +  PEEK (TL + 2
117 :
118  REM *** THEN PEEK/PRINT STRING
119 :
120  FOR X = 0 TO LE - 1
130 K =  PEEK (ST + X)
140  IF K > 127 THEN  PRINT CD$(K - 1
     27);: GOTO 160
150  PRINT  CHR$ (K)
160  NEXT
170  RETURN
```

Program 7.5.

Line-by-line analysis

Line 20: If you are using any variable, rather than entering a number (the C64 can find a variable value somewhat faster than it handles raw numbers), it's a good idea to 'initialise' them – that is, assign them a value – right at the start of the game. And if there's quite a lot of initialising and array filling to do, which all takes time, a good way to cover up the waiting time is to give the player something to read – extra instructions, a brief (one screen long) introduction to the first situation he will find himself in, etc. In this line we have to initialise the Base Address (BA) of the look-up table. Notice that the formula in line 100 *cannot* give a value lower than 3 (if PL=1 then PL * 3 = 3) so we must deduct that value from the starting address of the table so that the first byte it will look at is 1 (the first value of PL) * 3 + 20956 (BA) to get the real start of the table – at 20959 in this example.

Lines 100-110: The formula for TL (Table Locations) *must* be calculated every time we want to pull out a string for display. Having

found the address of the first of *three* bytes in the table we can now get the *length* of the string to be displayed (LE) and the STarting address of the POKEd string (ST). Since the C64 will always carry out a multiplication before it performs an addition, if they occur in the same line, we don't need to put brackets around PEEK (TL + 1) * 256 to tell the computer it must do this calculation before adding on the value held in TL + 2.

Lines 120-130: Note the calculation involved here. We must start collecting bytes from the address pointed to by ST and we must end at the last byte of the current room description. We could do this by writing in line 120 FOR X = 1 TO LE, which would be slightly shorter than the statement I've written. The trouble is that if we used this formula line 130 would start PEEKing at the *second* byte of the string, and end up on the first byte of the next string up. To avoid this we would either have to subtract 1 from the value calculated for ST in line 110, which is a possibility, or alter line 130 to read K = PEEK (ST − 1 + X), thus adding another calculation to every step of the loop. Which is not a very good idea at all! Line 130, by the way, is the one that saves so much time. Instead of having to search through a string for each successive letter and finding its ASCII code we already have either an ASCII value or a code number.

Lines 140-160: As before, we need to determine whether the value of K is above or below 127 to find out whether we're looking at a letter, to be printed direct to the screen in line 150, or a code letter, in which case we print out the appropriate group of letters and jump over line 150. Note that both print statements end with a *semicolon* to ensure that we print across the screen, not down.

Line 170: Lastly, having completed the subroutine, we can RETURN to the move handling routine to be sent on either to the 'event check' routine (see Program 6.1 notes) or to the command input routine, according to the method you have selected.

And talking of command input routines ...

Chapter Eight

The Well-chosen Word

The Inglish Language (that *is* spelt correctly) was invented for the game *The Hobbit* released by Melbourne House. The entire program was a team effort involving Melbourne's own managing director, Alfred Milgrom, who originated the eighteen-month project, and a team made up of Philip Mitchell and Veronica Megher (programmers), Stuart Richie (linguistics expert) and a pair of graphic designers who produced the artwork for both the screen displays and the packaging. But it is the presence of the linguistics expert which particularly interests me here. It emphasises the difficulty of programming a really effective 'command parser' – that part of the program which receives and interprets input from the player. It's worth looking at this area of the adventure program in some detail.

Sounds and words

If you have any interest in artificial speech units then you already know that the entire English language, in its spoken form, can be broken down into just 64 *phonemes* or sounds. The complexity of actual speech is due to the way that we link these sounds together, along with shorter and longer pauses, and the emphasis placed on each part of each word. On this basis it could be argued that our language is essentially rather simple, and that its complexity is largely dependent on the way it is used. Which is true, in a way.

Suppose I write something like: 'I want three more.' There's not a lot you can tell from this sentence on its own except that the speaker has already got at least one of something and wants three more. What you can't tell, just from *reading* the words, is whether the speaker is male or female, young or old, happy or sad. Neither can you tell whether the words are a command – 'I want three more (so

give them to me)' – or a request – 'I want three more (please?)'. The only way you could understand exactly what was going on would be to either *hear* the words spoken, or see the sentence in its proper setting. Now this book can't talk, of course, so you can't hear the sentence at all. But suppose I rewrite the sentence like this:

> The bricklayer turned to the hod carrier and said 'I want three more.'

Even though I've only added nine words – which don't really say much in themselves – the whole meaning of my original sentence becomes much clearer. I don't even have to tell you that the bricklayer wanted three more *bricks*, because in this context the meaning of the words is implied by who said what to whom.

The point is this. While producing artificial speech might seem very difficult, it's actually quite simple once you have the right hardware and the software needed to drive it. Indeed, I have a home-brewed program for the APPLE II+ which actually digitises recorded speech, stores the result in RAM, and plays the message back again simply by 'toggling' the speaker whenever it finds a bit which has a different value from the one before. The result wouldn't win any prizes for elecution, I admit, but it is possible to store simple messages like 'Hello. I am an APPLE computer', plus the record/playback program, in about 3.2K!

If only parsing written commands were that easy! But it isn't. In order for the computer to execute a command it needs to know not only what was *said*, but also what that particular collection of words *meant*. Just how difficult it is to extract the meaning from any command will depend upon the length and complexity of the command itself.

How many words make a sentence?

If we weren't talking about computing then this question would make about as much sense as the old favourite 'How long is a piece of string?'. Since we *are* talking about computing it makes very good sense indeed.

The original adventure programs, despite being run on mini-computers with around 256K or more of RAM, only accepted one- or two-word commands. In practice the one-word commands were restricted to directional commands – so GO EAST could be abbreviated to EAST, or even plain E. All other commands were

made up of two words, a verb and a noun, thus:

GET SWORD
THROW ROPE
DROP CANDLE

and so on.

Long or short?

This leads to a very interesting and fundamental question which has, as yet, received little or no public discussion. Is the implementation of complex parsers, necessarily a mark of progress in the field of adventure games?

I have argued, elsewhere, that the more commands can be made to resemble standard English the more realistic the game becomes. At the same time I am well aware that many players find it tiresome to enter every command *in full* – especially when they are re-running a game for the nth time. So which is really *best*? Frankly I don't know, and for that reason this chapter covers both alternatives.

It is, of course, possible to create an extremely simple command parser based on just one letter for each action. But this limits your choices quite considerably (one action for each keyboard character) and requires that the player keep looking up the instructions to find out which letter stands for which command. In other words, although games are still on sale which use the 'one key command' system, no one has yet produced a really satisfactory implementation (can you?!).

Program 8.1: Two by two

The next step up from the one-letter command is – you've guessed it – the *two*-letter command. This may not sound like much of an improvement, yet it does mean that (using capital letters only) we can implement a total of 676 commands. That's quite a step!

Like most systems the two-letter method does have its faults – the main one being that the player still has to learn which letter stands for which verb and noun. In the case of the verbs, initial letters are *not* the most obvious answer as one would have no way of distinguishing between, say Go West and Get Wood or between Drop Sword and Draw Sword. And of course the same problem applies in relation to the objects, which would each need a different initial letter in order to avoid confusion between Get Axe and Get Apple.

However, the problem is by no means insurmountable and a little re-definition can work wonders:

Get becomes Take
Go becomes Move (or Run, Walk, etc.)
Apple becomes Fruit

and so on.

So, we have two lists – verbs and nouns – but how do we access them? Actually the process is pretty simple, using a short formula given below, and as you can see in this program the operation breaks down into just four stages:

(1) The program first takes the ASCII value for each of the two letters in CO$ and deducts 64 from each so that A = 1, B = 2, C = 3, etc. (**Note:** there must not be a blank space between the two letters.)
(2) If either of the characters in CO$ is not an upper case letter then the 'DON'T UNDERSTAND' message is displayed and the program returns for a new command.
(3) When a command containing two 'legal' letters is found, the program moves on to a brief calculation based on the modified value of V (the first letter on CO$) and N (the second letter). From these two values, plus EN (a 'constant' which equals the total number of nouns *plus* one, i.e. EN=27) we can find a unique value for every command by using the formula:

X = ((V * EN) + N) − EN

Note that the second EN is used to make sure that the possible values of X start from 1 and not from EN+1. (If V=1 and N=1 and EN=27 then X=(V*EN)+N would produce the result X=28. In our formula EN is deducted again so where V=1 and N=1 and EN=27 the result of X=((V*EN)+N)−EN will be X=1.)
(4) Finally, having found a value for X, we can use the statement

ON X GOSUB 200,250,300 etc.

to move to the correct line to execute every possible command.

Just before we go on to the program itself I should point out that the use of this formula has one significant drawback – it allows all sorts of irrelevant pairings. In other words it is possible to link *any* verb with *any* noun. While you could certainly THROW a KNIFE, a ROCK or a ROPE it makes no sense at all to THROW DOOR or THROW WEST. So, while the formula method is highly efficient in one way, it does require that many of the addresses in the lines

ON X GOTO 200,250,300 etc.

will be directed to the subroutine which simply prints out 'I CAN'T DO THAT' and then goes back for a new command.

```
]LIST

1  REM ***** 2 LETTER CO$ PARSER  ***
     **
2 :
3 :
8  REM *** FILL VERB AND NOUN ARRAYS
9 :
10  DIM V$(10),N$(16):EN   17
20  FOR X = 1 TO 10
30  READ V$(X)
40  NEXT
50  FOR X = 1 TO 16
60  READ N$(X)
70  NEXT
77 :
78  REM *** AND SET VE$,NO$ AND BA$
79 :
80 VE$ = "ABCDEFGHIJ":NO$ = "ABCDEFGH
     IJKLMNOP"
90 BA$ =  CHR$ (20)
97 :
98  REM *** GET CO$ AS 2 ASCII VALUES

99 :
100  PRINT "[DOWN]WHAT NOW? ";
110  GET V$: IF V$ = "" THEN 110
120  FOR X = 1 TO 10
130  IF V$ =  MID$ (VE$,X,1) THEN VP =
     X:X = 10: NEXT : GOTO 150
140  NEXT : PRINT " ": GOTO 300
150  PRINT V$(VP);
160  IF VP = 2 THEN  PRINT " ": GOTO
     3000: REM  *** GO TO 'LOOK' ROUT
     INE
170  IF VP = 9 THEN  PRINT " ": GOTO
     3500: REM  *** GO TO 'INVENTORY'
      ROUTINE
180  GET N$: IF N$ = "" THEN 180
190  IF N$ = BA$ THEN  FOR X = 1 TO  LEN
     (V$(VP)): PRINT BA$;: NEXT : GOTO
```

```
      110
200   FOR X = 1 TO 16
210   IF N$ =  MID$ (NO$,X,1) THEN NP =
      X:X = 16: GOTO 230
220   NEXT : PRINT " ";: GOTO 310
230   PRINT " "N$(NP);
247 :
248   REM *** ALLOW CORRECTION
249 :
250 T = TI:T = T + 120
260   GET AL$: IF AL$ = "" AND TI    T THEN
      260
270   IF AL$ = BA$ THEN  FOR X = 1 TO
       LEN (N$(NP)) + 1: PRINT BA$;: NEXT
      : GOTO 180
280   PRINT " ": GOTO 400
297 :
298   REM *** DEAL WITH ILLEGAL CO$
299 :
300   PRINT "[DOWN]I DON'T UNDERSTAND
      [RVS]"V$: GOTO 100
310   PRINT "[DOWN]I DON'T UNDERSTAND
      [RVS]"V$(VP)" "N$: GOTO 100
397 :
398   REM *** PROCESS VALID COMMANDS
399 :
400 CO = VP * EN + NP - EN
410   ON CO GOTO
9997 :
9998   REM *** COMMAND DATA [DEMO ONLY
      ]
9999 :
10000   DATA GET,LOOK,CLIMB,DROP,EXAMI
      NE,THROW,GO,SAY,INV,WEAR
10010   DATA THE TREE,THE GLOVES,THE B
      ANANA,THE RING,THE SPADE,HELLO,Y
      ES,THE SWORD
10020   DATA THE BOOK,HEEBEE JEEBEE,NO
      RTH,SOUTH,EAST,WEST,UP,DOWN
```

Program 8.1.

Wasn't that fun? Well, it wasn't that bad! But I did start off by talking about the kind of advanced parsing routines found in *The Hobbit*, Infocom games and the like. And that's what I want to go on to now.

Every little word ...

What the more sophisticated command parsers are actually designed to do is to break every INPUT down into separate words, analyse each word – is it a verb, a noun, a preposition, etc. – and then go on to analyse the 'syntax' to make sure the command actually does look like good English. Now that's pretty heavy going, to say the least, and a major problem with trying to tackle a job like that – from our point of view – is that these routines are all written in machine code. In other words they execute a lot faster than any BASIC program trying to do the same job.

For BASIC programming, then, it is essential that we take a few short cuts in order to achieve an acceptable execution time. Whether these short cuts actually work will depend, to a large extent, on how closely we stick to the normal syntax of the English language. Fortunately this isn't too hard to do if we aim for the 'practical' approach:

(1) 'I the knife threw', is obviously not good English, even though it isn't too difficult to understand what the sentence means. 'I threw the knife', on the other hand, is perfectly good English, even though we've only moved one word. And if we modify it again to get a command rather than a statement – 'throw the knife' – we have the basis for our first rule: verbs must *always* come before the nouns they refer to. In fact, for adventure programs, we can insist that the first word of each command must be a verb. And if the first word of any command is not one of the program's listed verbs then the whole command string is declared invalid.

(2) Following on from the last rule we can also state that every verb must have a noun (though every noun doesn't have to have its own verb – more of that in a moment). 'Go quickly' may be good English, but the computer can't 'go' anywhere unless you give it a direction. On the basis of this second rule we can, at least temporarily, handle command strings by looking for a *valid* verb in the first place and then going straight off to find the noun that goes with it. For the time being we will ignore everything between these two words, including the syntax of the command.

Taking things this far, we're obviously still working with a system that looks very much like the old two word parsers of yesteryear, even though the input itself can be quite complex. The next step, then, is to broaden the system so that we can begin to deal with compound sentences.

(3) A compound command is, as you probably know, one which contains several different instructions in one INPUT. In order to separate the various commands we must introduce what are called *delimiters*, or 'boundary markers'. For all practical purposes we can restrict ourselves to the use of just five delimiters – the words 'and' and 'then', the punctuation marks ',' and '.', plus RETURN. Nevertheless, the use of *any* delimiters raises a whole nest of problems.

In the first place only a full stop or the RETURN key act as 'absolute' delimiters. *Then, and* and commas can all function as both absolute and secondary delimiters. For example:

GET THE AXE, GO NORTH
GET THE AXE, THE SWORD AND THE ROPE

or

GO WEST THEN EAST

For the sake of simplicity, then, we'll introduce a couple of subsidiary rules:

(a) Where a full stop is not followed by RETURN it must be followed by a verb.

(b) The words THEN and AND, plus commas, must be followed by a verb or a noun (the word 'the' counts as part of a noun). I'm not saying this is absolutely faithful to the rules of English grammar, as this sentence proves, but it's completely adequate for our purposes. This still won't solve all our problems, however. There are at least two more situations that need to be taken into account. Fortunately both of them are fairly straightforward.

(4) Suppose that we have more than one object of the same general description. How do we know which object the player wants to deal with? Is it the RED APPLE or the GREEN APPLE, the LONG SWORD or the SHORT SWORD? The simple answer would be to avoid the problem by never duplicating items. And this is also probably the best solution from the view of actually writing a program. But there may be times when you particularly want to introduce at least two similar items – to cause confusion, perhaps. So we'll deal with such situations by introducing another rule: the adjective for any noun must come immediately *before* the noun that it qualifies. Thus

THE LEFT DOOR

is valid, whilst

THE DOOR ON THE LEFT

though nearer to everyday English usage, wouldn't be allowed. I'm sorry if this part of the rules is a bit clumsy, but the reason for it will become abundantly clear when we get to the program itself.

(5) What, you may ask, about qualified actions? In two-word (or two-letter) commands one types in

KILL GOBLIN

the computer asks

WHAT WITH?

and you reply

THE AXE

or the sword, the mace, the one-ton jar of marmalade, or anything else you can lay your hands on.

Using our new super compound parser you might think we now have to deal with a whole new group of words, words like WITH, ON, UNDER, FROM, etc. Actually the problem never arises. Instead we follow the same logic as the two-word parser - we get the verb and the noun and then the program to execute that command

KILL GOBLIN

The difference comes when the program wants to know *what* we're going to kill the Goblin with. Instead of going back to the player and asking for more INPUT we go on through the original command string until we come to another noun. If a satisfactory noun is found, then the command is acted upon. If no noun is found that would fit the situation then we print a polite rejection, cancel the whole command string and go back for a new set of instructions.

Program 8.2: Compound Parsing

At this point I'd like to go on to the Compound Parse program itself, rather than get bogged down in too many details which I hope, will become a lot clearer when you see how they are dealt with in practice.

```
]LIST

1  REM *****  COMPOUND PARSER  *****
2 :
3 :
8  REM *** SET UP WORD ARRAY
9 :
10  DIM CO$(26),A$(30):VT = 10:NT   2
     6:EN = 17
20  FOR X = 1 TO 26
30  READ CO$
40  NEXT
496 :
497  REM *** GET CO$ AS A SET OF
498  REM       WORDS AND SYMBOLS
499 :
500 X = 1:A$(X) = "": PRINT "]DOWN]WH
     AT NOW? ";
510  IF X > 1 AND (A$(X - 1) = "" OR
     A$(X - 1) = " ") THEN X = X   1
520  GET A$: IF A$ = "" THEN 520
530  IF A$ =  CHR$ (20) THEN  PRINT A
     $;:A$(X) =  LEFT$ (A$(X), LEN (A
     $(X)) - 1): GOTO 520
540  IF A$ =  CHR$ (13) THEN  PRINT "
      ":CH = 1: GOTO 600
550  IF (A$ = " " AND X > 1) AND (A$(
     X - 1) = "," OR A$(X - 1) = ".")
      AND A$(X) = " " THEN  PRINT A$;
     : GOTO 510
560  IF A$ = " " THEN  PRINT A$;:X =
     X + 1:A$(X) = "": GOTO 510
570  IF A$ = "," OR A$ = "." THEN X =
     X + 1:A$(X) = A$: PRINT A$:X = X
      + 1:A$(X) = "": GOTO 510
580  PRINT A$;:A$(X) = A$(X) + A$: GOTO
     520
597 :
598  REM *** VERB SEARCH
599 :
600 W = CH:WF = 0
610  FOR VS = 1 TO VT
620  IF A$(W) = CO$(VS) THEN VP = VS:
     VS = VT: NEXT :W = W + 1: GOTO 7
     00
```

```
630  NEXT
640  PRINT "[DOWN][RVS]COMMANDS MUST
     START WITH A VALID VERB.[OFF]":V
     S = VT: NEXT : GOTO 500
697 :
698  REM *** NOUN SEARCH
699 :
700  FOR NS = VT + 1 TO NT
710  IF A$(W) = CO$(NS) OR A$(W)     RIGHT$

     (CO$(NS), LEN (A$(W))) THEN NP
     NS:NS = NT: NEXT : GOTO 800
720  NEXT
730  IF W < X AND WF    0 THEN W = W +
     1: GOTO 700
740  IF WF THEN 600
750  PRINT "[DOWN][RVS]IF YOU'VE INCL
     UDED A VALID NOUN I SURE  CAN'T
     FIND IT!![OFF]": GOTO 500
797 :
798  REM *** PROCESS COMMAND
799 :
800 CO = VP * EN + NP - EN
810  ON CO GOTO
1997 :
1998  REM *** CHECK ADJECTIVE
1999 :
2000  IF  LEN (A$(W)) <  LEN (CO$(NP)
     ) THEN B$ = A$(W - 1) + " " + A$
     (W)
2010  FOR Q = VT + 1 TO NT
2020  IF B$ = CO$(Q) THEN NP = Q:Q =
     NT: NEXT : GOTO 2040
2030  PRINT "[DOWN]SORRY - THERE IS N
     O [RVS]"A$(W - 1)"[OFF]"A$(W): GOTO
     500
2040  PRINT "[DOWN]O.K.": END
2050  IF W < X GOTO 20000
9997 :
9998  REM  *** CO$ DATA
9999 :
10000  DATA GET,GO,TAKE,READ,DROP,OPE
     N,UNLOCK,ENTER,THROW,KILL
10010  DATA EAST,E,WEST,W,NORTH,N,SOU
     TH,S,DOOR,WINDOW
```

```
10020  DATA GREEN BOOK,KEY,ROCK,KNIFE
     ,GOBLIN,HAMBURGER
19997 :
19998  REM *** CHECK NEXT 'WORD'
19999 :
20000 W = W + 1: IF W < X AND (A$(W) =
     "THEN" OR A$(W) = "," OR A$(W) =
     ",") THEN 20000
20010  IF W < X AND (A$(W) = "THE" OR
     A$(W) = "AND") THEN 20000
20020  IF W < X THEN CH = W:WF = 1: GOTO
     700
20030  GOTO 500
```

Program 8.2.

Line-by-line analysis

Given the length of this program you may well be wondering what all the fuss was about. After all you'd hardly call it a mammoth piece of coding. This is, in fact, due to the situations I've chosen to deal with and those I've chosen to ignore. Besides, I couldn't see much point in creating a program which was marvellous at sorting out the meaning of the input but took five or ten minutes to handle each command. Here, then, are the line notes for the program:

Lines 10-40: The CO$() array is used to contain the set of words which are actually recognised by the parser and will be filled from the DATA statements in lines 10000-10020 in lines 20-40. The A$() array is used to hold each set of commands as a series of separate words. The size of the array takes account of the normal size of input that the C64 will allow, and exercises what I think is a reasonable limit on the number of instructions that can be given at any one time. VT, which stands for Verb Top, gives the length of the verb section of CO$() – see lines 600-640. NT (Noun Top) gives the element of CO$() which holds the last noun. EN holds the value for total number of nouns plus 1 for the formula in line 800.

The next section of the program is essential to its success. It must be entered very carefully, taking especial note of the positions of the brackets: one misplaced bracket could well crash the whole routine. Because these lines are so important I want to deal with them in some detail.

Line 500: This routine actually has three entry points, of which this is the first. Before any *new* command input is accepted, X must be reset

to 1, and the first element of A$() is cleared to become a 'null string'. The familiar "WHAT NOW?" enquiry is then displayed.

Line 510 is the second entry point into this routine, used whenever a word or symbol has been accepted as part of the A$() array. Its purpose is to ensure that no blank spaces or null strings are accepted as part of the final version of A$() when we move on to the analysis routines.

Line 520: The third and final entry point, this line actually collects the individual letters and symbols for the elements of A$(). If we used INPUT rather than GET then commas would not be accepted *by the computer* as legal material.

Line 530 deals with deletions made during input. The deletion is made both in the screen display and in the element of A$() currently being entered. *But beware*, this line cannot tell the difference between a *deletion* and an *insertion.*

Line 540 comes into play when the RETURN key is pressed. The input string is terminated, the value of CH is reset to 1, and execution moves on to line 600.

Line 550 handles any blank spaces entered for A$, entered after a comma or a full stop. In either of these situations execution moves to line 510, which will clear the relevant element of A$() to prepare it for fresh input.

Line 560: When a blank space appears at the end of a word this line ensures that only the word itself enters A$(). The value of X is then incremented and the next element of A$() is cleared to prepare it for input. (**Note:** Although they are not included in A$(), all blank spaces will appear on the screen.)

Line 570 handles commas and full stops *only* as valid input, assigning them to their own elements of A$(). Any other punctuation marks would, unless deleted at the time of entry, be included as part of the preceding word. Finally line 570 handles any other input, either letters, numbers or symbols, adding them to the current element of A$(). Note that only this line and line 530 (deletions) return immediately for a new character without having line 510 check for unwanted blanks. So don't include blank spaces in words unless they are meant to be there.

Line 600: The variable W indicates which word in the A$() array is being handled at any moment. If a new command input is being

handled then W will be set to one – from the value given to CH in line 540. Other situations, when we're looking at a new command rather than a new input, are dealt with in line 20000. Likewise the value of WF is cleared to 0 to show that we are looking for a new command – see lines 20000 and 730.

Lines 610-630: On the basis that every command *must* start with a verb this loop only searches the first VT elements of CO$ for a match for the first element it can see in A$() – element A$(W). If A$(W) *is* a verb execution moves on to line 700, otherwise ...

Line 640 prints a carefully worded rejection and refuses *all* the current command input. This may seem a bit drastic, but it does save the player from finding that, for example, because they weren't able to GET THE SWORD they now have to try to KILL THE GOBLIN with their bare hands.

Lines 700-720: A simple loop which searches through A$() for a noun. Here I've allowed this search to include a match with the last LEN(A$(W)) letters of any of the nouns to cope with elements which include an adjective for clarity (see GREEN BOOK in line 10020).

Line 730 ensures that this search continues until we find a noun, reach the end of A$(), or ...

Line 740: ... WF has been set in line 20000. In this case what we're looking for is a new command, so program execution is sent back to the start of the *verb* search routine. If this is still a bit confusing, don't worry. All will become clear when we get to line 20000.

Line 750: If none of the above conditions results in a satisfactory pairing of a verb and a noun then all of A$() is scrapped (as in line 640), an error message is generated, and the program goes back to line 500 for fresh input.

Lines 800-810: A repeat of the 'unique commands' formula from lines 400-410 of the two-letter command parser.

Lines 2000-2040: This subroutine is designed to find the adjective that accompanies a given verb – remember my example of THE LEFT DOOR? When we left the noun search routine we did not increment the value of W. This means that A$(W) will still be the noun of the command we wish to execute. But supposing it is not the *whole* of the noun as given in the CO$() array because it was let through by line 710? Here we look back at the previous word – A$(W–1) – attach

it to A$(W) with an intervening space, and search for a match in the noun section of CO$(). If we still cannot find an exact match then an error message is generated, the rest of the command input is scrapped and the program returns to line 500.

Lines 2040-2050: If we now have a satisfactory match then the command can be executed and the program should be sent to line 800. Line 2040 should actually appear at the *end* of each command execution routine as its function is simply to discover whether we've used up all the words in the command string held in A$() – is the last word of the current command (W) the last word of the command input (X)?

To see this routine change line 800 to GOTO 2000 and then RUN the program using the command

GET ME THE RED BOOK

Lines 10000-10020: Contain the verbs and nouns to be stored in CO$().

Lines 20000-20030: We're there at last! So what does this second subroutine do?

Firstly line 20000 adds 1 to the value of W, taking us on to the next word of the command string, and then checks what the next word is. If there is a next word, and it is not a recognised delimiter, then the value of CH is reset to the current value of W so that the word can be recognised as the first word of a command in lines 600-640 – *if necessary*. Then we set the WF 'flag' to 1 before sending the program off to do a noun search. Yes, a *noun* search. Because we might be looking at a command like

GET THE SWORD AND THE HAMBURGER

or

GET THE KNIFE, THE KEY AND THE ROCK

in which case we need the *same* verb with a new noun. If we're wrong about this, and we're actually dealing with a brand new command, then setting WF to 1 ensures that a failed noun search will be followed by a verb search.

Laying your cards down

And so we come to another of the unresolved questions in adventure

gaming: Should the writer document the words that can be used to make up valid commands, or should the players be left to find out for themselves?

My own feeling is that a game should contain worthwhile problems for the player to solve without adding the need to discover what words make up the writer's own game vocabulary. In defence of this view I would cite The Hobbit and Valhalla, which not only give word lists in the instruction booklets but also quite lengthy explanations of how the words may be put together to form legal commands. Admittedly Infocom instructions are not quite so detailed in this respect, but then a vocabulary of 600 words or more would probably do more to confuse the player than help him.

The one exception to this view which I do find acceptable is the note in the handbook for The Hobbit which explains that a few special words – actions and nouns – have been omitted from the list, though it should become apparent during the game what these words might be. Certainly the writer should not be obliged to give away all his or her secrets in advance, but it is a good idea to keep such special cases down to a minimum. Nothing can be more frustrating for a player to find that he cannot manage a perfectly obvious action because the writer has not only failed to disclose his basic word list, but has occasionally chosen words which are themselves by no means obvious. For example, it may come quite naturally to one person to talk of *scaling* a wall, so that when a player uses the more mundane, but more widely accepted (?) command CLIMB THE WALL the computer replies YOU CAN'T DO THAT. Should the player take this to mean that there is no way over the wall, perhaps most people's first reaction, or does he run down to the local bookshop for a copy of Roget's *Thesaurus* so he can try out every possible variation on the word CLIMB?

In broad terms, then, I would argue that, unless there is a very good reason indeed for *not* making a word known – such as concealing the existence of a special object until it is found in the course of the game (because the player would almost certainly guess what it was for if he knew it existed) – then all valid words should be disclosed in the documentation which accompanies the game.

Chapter Nine

Taking Shape

At this point in the book it's worth taking some time out to consider the subject of program organisation - Block Diagrams, Flow Charts, etc.

Whenever people talk about 'organising' programs they inevitably think of *structured* programs. This word 'structured' - as in Structured BASIC, Structured Programming and so on - is one that is frequently used, yet seldom explained. Many people think it is 'a good thing' to use structured programming, even if they aren't too sure what it really is.

Take this example from a computing magazine:

> 'Structured programming is *not* splitting up a program into a sequence of subroutines, each one performing one task as a part of the whole process.
>
> 'It is *not* the use of constructs such as WHILE... or IF...THEN...ELSE... statements in a language.
>
> 'It is *not* the elimination of all unconditional jumps (e.g. GOTOs) from a program.
>
> 'It is *not* the ability of a language processor to produce auto-indented listings of programs.
>
> 'It is *not* the provision of labels, named subroutines, data declaration (local or global).'

Well, that's what structured programming *isn't*. So what *is* it?

The essence of a structured program is that it shows good organisation. As one programming guide explains (referring to Structured COBOL):

> '... one of the major advantages of structured programming is that it is readable by human beings. A person who isn't a programmer at all can read a structured COBOL program and figure out what is happening.'

And what is true for a program written in COBOL should also hold true for a program written in BASIC (given that BASIC is not as 'high' a language as COBOL, relying to a greater degree on symbols and mathematical constructs).

The term 'structured' seems to have arisen from the spread of computer software into the 'outside' world where it was more and more likely to be used by people with little or no knowledge of computer languages or programming methods. A well-structured program is one that is clearly laid out and therefore (a) can be debugged relatively easily and (b) can be read, understood and modified at a later date by someone other than the original programmer(s). So the pursuit of structure is not merely an academic whim. It has a worthwhile, practical purpose.

It is not the purpose of this chapter to provide a complete course in structured programming. What I will try to do, however, is to provide some useful pointers to better - by which I mean more effective - programming methods.

Block diagrams

The task of preparing the actual program for an adventure game follows much the same pattern as we saw when we were preparing the plot and storyline for a game. The main difference is that we now know what we *want* to do but still have to decide *how* it is to be done. A useful first step in organising the actual 'encoding' of a program is to prepare a *block diagram* (sometimes known as a *systems flowchart*).

Drawing up a block diagram actually has two purposes. Firstly it is an easy way of sorting out exactly *what* is to go into a program. (Easy to do, and easy to read afterwards.) Its second function is to show the relationships between various sections of the program within the overall setting. To put it simply, the block diagram shows *what* goes *where*.

Let's lay out a sample diagram (Fig. 9.1) and you'll see what I mean.

(**Note:** in this example I've assumed that the main Introduction to the game and the Character Generator will be LOADed and RUN prior to the LOADing of the main program.)

Obviously what I've drawn here represents a very bare minimum. It doesn't cover any of the details of the game except - as briefly as possible - the handling of various types of INPUT (the player's commands). Indeed, this diagram could apply to almost any

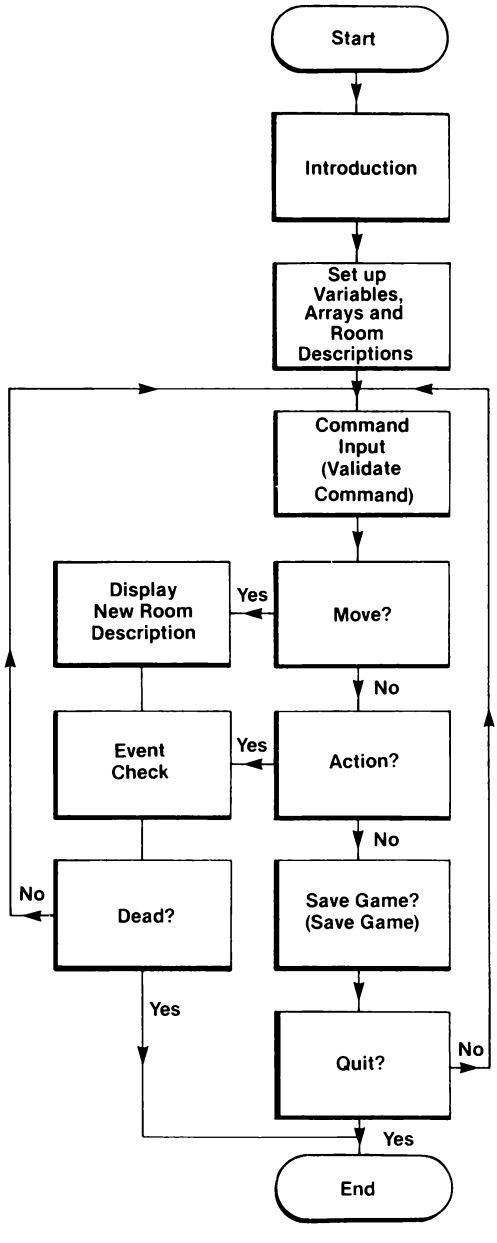

Fig. 9.1. Block diagram.

adventure game. Which is exactly how it should be. For while the flowchart for each game will be unique (and may need substantial alteration before it reaches its final form), the block diagram is for guidance only, and may not need to be altered through several different games. In fact it would only need to be changed if some quite new area of operation were introduced – the use of graphic displays, for example – or if you wanted to remind yourself how a radical new subroutine fitted into the rest of the program.

Having defined the main areas of the program, we can now begin to turn this into something rather more concrete. The first step will be to set aside specific areas of the program for specific tasks – the areas are defined in terms of line numbers, of course. Even if you intend to produce a final program with entirely consecutive line numbering (thus making it harder for 'intruders' to sort out individual sections of the program) it's still well worth breaking the earliest versions of the program down into clearly defined modules. Not only does this make the process of de-bugging each program section much easier, it also allows you to save individual modules for inclusion (as they stand or in modified form) in future programs.

The second step is to prepare a detailed or *program* flowchart (see Fig. 9.2).

Little boxes

'Why bother to prepare a flowchart in the first place? By the time you've finished laying out a chart you might just as well have written the program itself.'

This is a common argument, and one that sounds particularly convincing when you've just altered a chart for the tenth time to include a subroutine that had previously gone unnoticed. Unfortunately it isn't a very accurate argument. As someone once said, ninety out of every hundred programmers *cannot* put a program together successfully without first preparing an adequate flowchart. And of the other ten, nine only *think* they work well without a chart.

Is this an over-cynical attitude? I don't think so. After all, there's a lot more to the flowcharting process than just stringing together a variety of little boxes. First there's the question of how you will set up various sections of the program – the actual construction of lines of BASIC. If you write your program 'at the keyboard' then, unless you're a very experienced programmer, there's a strong likelihood that you'll type in whatever seems good at the time. And as long as it

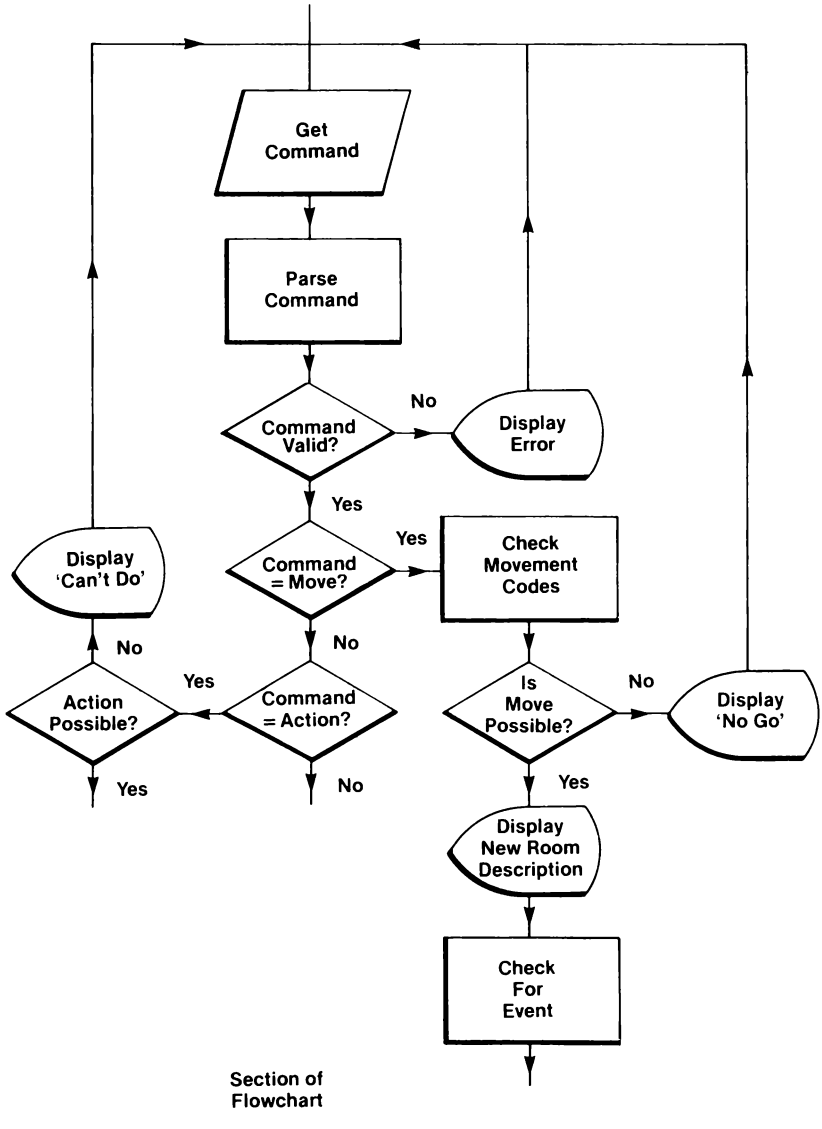

Fig. 9.2. Flowchart.

works it's liable to get left just the way it was written - even if you come up with a better version of the routine, or even a whole new approach. To put it bluntly it's a lot easier - physically and emotionally - to change a piece of code that you've roughed out on paper than one that's already been entered, tested and which stands ready to run.

IF C<10 THEN Q=V(R,C)

As if you hadn't already guessed it, we now come to the subject of variables. Broadly speaking, there are two types – 'local' and 'global'. A local variable is one which has relevance only within the subroutine in which it appears. A 'global' variable is one which remains active, so to speak, throughout the program. Controlling the use of your variables is a task which needs to be approached with considerable care.

There are some languages which allow the same variable names to be used both locally and globally without causing total chaos. Obviously this is extremely useful. It is equally obviously not an option allowed for by the C64's version of BASIC. It is down to you, the programmer, to keep meticulous track of what names are being used for what, where and when.

Now some people will argue that specific variables don't belong in a flowchart, and that it should indicate *processes* rather than snippets of actual code. This is, however, very much a matter of opinion, and it seems a bit daft to miss out on one of the most valuable functions that a flowchart can serve for the sake of standing on a rather dubious principle. Having got that off my chest, let's get back to the plot.

The best illustration of the use of a *local* variable is the way that X is used in most of the programs in this book. On almost every occasion where I have used X it is as the 'index value' for a loop. X is, then, a very definite local variable. It might crop up in many places in a program, but its value is constantly being re-set, seldom remaining the same for more than four or five lines of code. Because X is used in this way it has been necessary on a few occasions to 'skim' the value of X and to hold it in another variable for subsequent use while X is re-valued once again.

This process can be seen quite clearly in the two-letter command parsing routine (Program 8.1). In lines 110-130 X is the index value for a loop in which an INPUT string value is compared with a list of acceptable values for that INPUT. If a match for V$ is found in the longer string – VE$ – then the value of X is transferred to VP (as a index to the array V$()), and X is re-set to the highest possible value for that loop to allow an immediate exit.

In this context the V$() array and the checklist VE$ are *global* variables which must retain their initial values throughout the program. VP is a 'major' local variable in that its value normally only alters when the routine is called (though it will also be affected

by an incorrect INPUT). Poor old hard-working X is an utterly local variable which is re-valued just five lines after it has been set.

To put it another way, if either VE$ or any element of the V$() array were re-set during the RUNning of the program then the whole program would become inoperable (especially in the case of VE$). VP, on the other hand, while holding its value for the duration of the subroutine, could be used for another *short-term* purpose outside the subroutine, since it will automatically be re-set next time the parsing routine is activated. And X is not only used elsewhere but is actually re-set not once but twice within the one subroutine.

The second important aspect of flowcharting, then, is to ensure that the names and functions of all variables are defined in advance of the moment when you start to enter the BASIC code for your game. This avoids overlapping use of any particular variable *and* avoids having too many variables where one or two could be made to do the work of ten. After all, the Variable List Table takes up space as well.

The third major value of preparing a satisfactory flowchart is also concerned with economy. As I said before, it is all too easy, when preparing a program 'off the top of your head' to make routines unnecessarily complicated or even to write in different versions of the same routine where a single subroutine would do. The problem is that by the time you've spent two or three weeks preparing a program, you become so familiar with it that errors like this become increasingly difficult to spot unless they actually crash the program.

By laying everything out in chart form before you even touch the keyboard it's much easier to pare down and maximise the efficiency of your routines than when you're running through several pages of listing (especially if you don't have a printer and have to work from the screen).

The final touches

Well, that's about all of the really hard work dealt with. To finish off this chapter I'd just like to go over a few short points that will, I hope, be useful in improving your general programming technique.

(1) GOTO/GOSUB and REM

Even experienced programmers often address GOTO and GOSUB statements to a routine which starts with an explanatory REM line. This a *mistake*! Not long ago, for example, I came across an

American magazine where the editor had to apologise for a program in a previous issue. It included no less than twenty-eight occasions where REM lines – deleted before publication – had been used as target addresses! No doubt many readers had spotted the errors while entering the program in question and made the necessary alterations. But the fact remains that the errors were unnecessary, and should not have occurred in the first place.

Since REM lines are very useful in the early stages of preparing a program – particularly if you intend keeping a copy of the listing for the future reference – I would suggest that all REM statements are set out on separate lines from the main program, with 'odd' line numbers that can be easily recognised and deleted from the final version.

(2) Loop breaking

'Thou shalt not break out of an uncompleted loop' has long been one of the cardinal rules of good programming. The reason for this rule – leaving aside the fact that anything else is messy writing, and messy thinking – is related to the way in which many computers handle a BASIC program.

A standard way of handling loops is to store the starting address of the operations *within* the loop on what is called 'the stack' – an area of RAM set aside specifically for this kind of task. Thus, whenever the interpreter reaches the NEXT command at the end of the loop, the address at the top of the stack is transferred to TXTPTR (or TEXTPOINTER – assembly language labels are traditionally limited to 6 characters even if the Assembler can handle more) *if* the loop is not complete, and the looped operations are repeated again.

In fact, as we've already seen, the C64 uses a rather different method of moving through a program, using a 'middleman' buffer. Thus TXTPTR actually points to the buffer area rather than to the main BASIC program area. Not surprisingly, then, we find that it also deals with loops rather differently. The start address is not stored on the stack but on the zero page at addresses 61 and 62 (decimal).

In theory, then, it is not possible to overload the stack by repeatedly jumping out of uncompleted loops. Nevertheless it is unwise to take the chance of crashing a program just because you believed it couldn't happen! The way to get out of a loop before it is complete (that is, before the loop indexer has reached its highest

value) is actually very simple - as can be seen in the subroutine below:

```
110 FOR X = 1 TO 100
120 IF A$ = B$(X) THEN TE = X: X = 100: NEXT: GOTO
    150
130 NEXT
140 PRINT A$" IS NOT VALID.": END
150 PRINT A$" = B$(";TE;"). ": END
```

Incidentally, the amount of time saved by using the last three statements in line 120 is actually quite significant. If a demonstration array is created for B$() in which A$ is equal to the 50th element then line 150 is completed in about .4 of a second (using three letter items for A$ and all of the array). If A$ is re-set to equal the 25th element in B$() then the execution time drops to .2 of a second. If no escape method is used then it takes .6 of a second to complete the entire loop. This may not sound too hot, but suppose you're using this routine over and over again (as in a command parser). Taking the examples above you actually save 33 and 67 per cent of the execution time needed for the complete loop. And as the comparisons grow longer so will the amount of 'real time' that you can save.

(3) Program layout

Every type of program has its own distinctive 'architectural style'. Many business programs, for example, are made to be used by people who have little or no knowledge of computer programming and functions. Because of this they often have an opening sequence made up of one or more menus, each followed by a set of conditional GOSUBs.

Adventure programming also encourages a certain style of layout influenced by the essential constituents of such programs. This layout can be broken down into the following list of general routines:

(a) The Introduction
(b) Set up room descriptions and other variables, arrays, etc.
(c) The most frequently used subroutines (preceded by a jump to the main program)
(d) The main program
(e) A body of routines to deal with specific (usually 'one off') situations
(f) The End of Game routine
(g) DATA statements

I've already expressed my own preference for RUNning the Introduction and Character Generator as a separate module, so this leaves us with units (b) to (f).

Generally speaking any adventure program should start off looking something like the layout I've described. This allows us to chop off DATA at the end of the program once it has been stored elsewhere. But why put subroutines at the *beginning* of the program?

The reason for this is related to the way in which the C64 (and many other micros) finds the line it has been sent to by a GOSUB or a GOTO instruction. In order to find such lines the computer goes back to the very start of the program (pointed to by locations 43 and 44, using the formula PEEK(44)*256+PEEK(43)) and then scans through the BASIC program area until it finds the target line number. On the RETURN journey, however, it goes directly to the last byte of the GOSUB instruction and then moves forward one byte (which brings it up to a mid-line colon (:) or an end of line zero) and the next instruction is then dealt with. This is why you can jump from or RETURN to the middle of a line, but can only jump *to* the start of a line.

(4) Beating the intruder

A secondary benefit of using assembly language (or machine code) rather than BASIC is that low level language programs are much harder to decipher unless you are already familiar with the program. However BASIC, too, can be rendered at least semi-incomprehensible if you wish to make it so.

It may seem rather weird, having argued in favour of well-organised programs and flowcharts, to suggest that anyone should deliberately jumble a program up. Yet a well-organised program is easier to scramble than one that is none too clear in the first place. The point is that all writers/programmers have a certain 'style' – even the bad ones. Thus anyone who wants to break into a program has only to search out that overall style and they will soon be able to follow the flow of the entire program.

If, on the other hand, you start out with a well-defined set of modules it is relatively easy to rearrange them in a fairly random fashion that has nothing whatever to do with your normal method of working. In this situation an intruder will have to track down every variable, every GOTO and every GOSUB before they can begin to get any clear picture of *how* you are doing what you are doing.

Obviously the final result of such manoeuvres will never have the same air of inscrutability as as unexplored machine code program,

but it is fairly certain that less experienced 'peekers' will soon abandon the struggle.

(5) Speed tips

(a) Define *all* frequently used numbers as variables at the start of the program. The computer can actually find a variable in the VLT faster than it can evaluate and use raw numerical data.

(b) Try to arrange your variables in order of priority. Since the computer always starts reading the VLT from the top (that is, in the order that variables have been defined) the most frequently used variables should be at the top of the table.

(c) Don't use integer variables (such as A% or VC%) unless you really need to. In order to handle an integer value the C64 turns it into a floating point value and then back again. In fact the only reason for using integer variables is to create an integer array, as this does save three bytes per element over 'real' arrays.

(d) Don't bother to define the index variable in a loop – use NEXT rather than NEXT X or NEXT VC. This saves on the time that the computer takes to see if it's NEXTing (is there such a word?) the right loop. So long as you haven't incorrectly *nested* your loops (i.e. as long as they don't overlap) the computer will execute each loop correctly all by itself.

(e) Use single-letter variables wherever you can. In a long program you will almost inevitably need to use some two-letter variables, but it does take time to handle that second character. If you do need to use both one- and two-character variables, then try to make sure that the one-letter variables are assigned to the most frequently-used values. The time saved in one operation may be negligible, but when a routine is called tens or even hundreds of times during the program the overall saving can be quite significant.

(f) And finally – don't forget to remove all REM and 'spacer' lines from your final program. It's always a good idea to keep a back-up copy of your programs so why not SAVE the REM'd version, delete all unnecessary material, and then SAVE the compact version as your final copy.

By the way, do make sure that you've really completed the program before you do this, or every alteration will need to be duplicated, a boring task at the best of times and far worse if all your work is on tape.

Chapter Ten
Sound and Vision

There will, I'm sure, come a day when adventure programs without graphics of some sort come to be regarded as out-of-date. At the same time I think, for reasons which will be explained in the last chapter, that the time for this is still several years in the future. In the mean time it's worth asking to what extent we can introduce sound and graphics into adventures written for the C64 *to our advantage*. My own first reaction to this question – when I was roughing out this chapter – was that there really weren't many advantages involved in going beyond pure text.

'Hang about,' says the voice (I wondered where he'd got to!). 'They've got *The Hobbit* on the C64. And that's just for starters!'

Which is true. But it also highlights the old theme of BASIC versus Machine Code. The Commodore 64 is so named because it includes a whole 64K of RAM space. And it does – but not in an easily accessible form. For not only is a 2K chunk of the 'easy-to-get-at' RAM already allocated to the computer – and thus rendered unusable for all practical purposes – but much of the other RAM space that *appears* to be free isn't! Not as long as you're using BASIC. The way this paradox comes about is as follows.

When you first switch on your machine the title block at the top of the VDU screen announces that you have '38911 BASIC BYTES FREE'. Given that 1K actually equals 1024 this means that we can use just a fraction under 38K of RAM. So what happened to the other 26K?

Part of it, we know, is computer-used space – pointers, the stack, the BASIC buffer, etc. But even after deducting the 2K so used there's still 24K unaccounted for. The answer is to be found in the C64's method of mapping the memory. If you can get hold of a good memory map you'll see that most of the addresses above 40960 (the top of BASIC program space) are actually listed twice. Thus everything from 40960 to 65535, with the exception of the block

from 49152 to 53247, is taken up by SID. VIC, BASIC ROM, Kernal ROM and so on. But if the right *bits* of address 1 on the zero page are set correctly then this whole area is converted to RAM. What actually happens is that since the same set of addresses are shared by two separate sets of chips, the zero page address tells the processor which set of chips it should deal with at any given moment.

Now, if all you want to do is to POKE data *into* this high area of RAM then the computer automatically treats it as RAM. But just you try to get anything back out while you're using BASIC. What you'll PEEK at is the contents of the ROM, SID and VIC chips!

The problem is, of course, that the processor *needs* the ROM in order to execute the BASIC instructions. And since it can't read two versions of the same address at the same time it will only read the set it needs. Of course you could switch the address pointer over yourself by POKEing 1 with the appropriate number. But your BASIC program would immediately cease to function. In other words you couldn't recover data from that area of RAM even if the processor was looking at the right set of chips! Frustrating, isn't it?

So, apart from the 4K block from 49152 to 53247, the RAM between 2048 and 40959 (plus a few small sections of the area below BASIC) is *normally* all that we have to play with. And on top of this we must remember that (a) the 4K referred to can only be used for storage, not programs, and (b) if we try to use high resolution graphics we'll lose a sizeable chunk of the normal BASIC program area. And that brings us back to the all important question. How much space can we really afford to use up for sound and graphics routines?

My own feeling is that in a BASIC adventure any graphics element needs to be economical, extremely relevant *and* well designed before it is even worth thinking about. *Good* graphics may help to enhance a well-written game, but even the best graphics the C64 can manage (and they're pretty good) won't do much for a poorly-written game. (Except, perhaps, to make it look even worse by comparison.) And the same thing goes for sound routines, be they musical accompaniment or sound effects.

Having said that, is also occurred to me that where the *Introduction* to a game is LOADed and RUN before the main program is loaded into the computer, there would be room for something a bit special in the way of scene-setting, where anyone with the necessary artistic talents (and patience) could afford to let their hair down.

I mentioned patience there because of the limitations of the C64's built-in BASIC. Unfortunately, due to the absence of the necessary BASIC commands, both graphics and sound – though extremely well catered for in terms of hardware – are none too easy to use. Rather than try to do them justice in a few short pages, I've chosen to limit the discussion in this chapter to relatively brief descriptions of the VIC and SID chips and how to *begin* to use them.

Program 10.1: Looking spritely

There was a time, B.C. (before Commodore), when everyone knew them as MOBs – Moveable Object Blocks. But the full name was a bit of a mouthful, and the shortened version sounded positively barbaric, so they were given a lovely new, utterly harmless-sounding name. *They* are, of course, *sprites*. And it's with sprites alone that I want to deal in this first section.

Since the best way to find out how to do things on a computer is by actually *doing* them I've included a 'map' of the Vic chip (see Table 10.1) plus what probably looks like a rather simple program. It is, in fact,

VIDEO INTERFACE CHIP (VIC)
Base Address (=BA):53248
Top of VIC: 53294
Size in BYTES: 46
Pointers for SPRITE data: 2040-2047
(Address of DATA for SPRITE X = PEEK(2040+X) * 64)

VIC CONTENTS

Absolute Address	*Relative Address*	*Contents*
53248	BA+0	Sprite 0 – X (horizontal) co-ordinate
53249	BA+1	Sprite 0 – Y (vertical) co-ordinate
53250	BA+2	Sprite 1 – X co-ordinate
53251	BA+3	Sprite 1 – Y co-ordinate
53252	BA+4	Sprite 2 – X co-ordinate
53253	BA+5	Sprite 2 – Y co-ordinate
53254	BA+6	Sprite 3 – X co-ordinate
53255	BA+7	Sprite 3 – Y co-ordinate
53256	BA+8	Sprite 4 – X co-ordinate
53257	BA+9	Sprite 4 – Y co-ordinate

Absolute Address	*Relative Address*	*Contents*
53258	BA+10	Sprite 5 - X co-ordinate
53259	BA+11	Sprite 5 - Y co-ordinate
53260	BA+12	Sprite 6 - X co-ordinate
53261	BA+13	Sprite 6 - Y co-ordinate
53262	BA+14	Sprite 7 - X co-ordinate
53263	BA+15	Sprite 7 - Y co-ordinate
53264	BA+16	Where the X co-ordinate is greater than 255 then this location holds the value 1 and the Sprite X register contains X co-ordinate - 256.
53265 to 53268	BA+17 to BA+20	These locations are not to be used by the Sprites
53269	BA+21	'Enable' each Sprite by setting the appropriate bit to 1
53270	BA+22	Not used by Sprites
53271	BA+23	Set appropriate bit to double Sprite height
53272 to 53274	BA+24 to BA+26	Not used by Sprites
53275	BA+27	If appropriate bit is *clear* (=0) then Sprite will appear in front of background. If set (=1), Sprite will appear behind
53276	BA+28	Set appropriate bit for 'multi-coloured' Sprite (see 53285 and 53286)
53277	BA+29	Set appropriate bit to double Sprite width
53278	BA+30	Computer sets appropriate bits if two Sprites 'collide'
53279	BA+31	Computer sets appropriate bit if Sprite 'collides' with background
53280 to 53284	BA+32 to BA+36	Not used by Sprites
53285	BA+37	Second colour (0-15) for multi-coloured Sprite(s)
53286	BA+38	Third colour (0-15) for multi-coloured Sprite(s)
53287	BA+39	Primary colour (0-15) for Sprite 0
53288	BA+40	Primary colour for Sprite 1
53289	BA+41	Primary colour for Sprite 2
53290	BA+42	Primary colour for Sprite 3
53291	BA+43	Primary colour for Sprite 4
53292	BA+44	Primary colour for Sprite 5
53293	BA+45	Primary colour for Sprite 6
53294	BA+46	Primary colour for Sprite 7

Table 10.1. Structure of the VIC (video interface chip).

nearly as simple as it looks, but it also serves to illustrate all the most important aspects of sprite manipulation. For those readers who already have a sound knowledge of this subject I would re-emphasise that this is only an *introduction* to sprite handling.

By the way, what the program actually does is to create a rather 'bouncy' horse which tracks backwards and forwards across the screen. The rest of the program details can be found in the line notes which follow.

```
]LIST

1  REM *****  SPRITE 0 DISPLAY  *****

2 :
3 :
8  REM *** SET SCREEN COLOURS
9 :
10  POKE 53280,5: POKE 53281,6
17 :
18  REM *** SET DATA POINTER
19 :
20  POKE 2040,13
27 :
28  REM *** POKE SPRITE 0 DATA
29 :
30  FOR SD = 832 TO 894
40  READ A: POKE SD,A
50  NEXT
100 V = 53248
110  POKE V + 27,0
120  POKE V + 21,1
130  POKE V + 39,0
140  POKE V + 39,0
150  POKE V + 23,1: POKE V + 29,0
160  POKE V + 1,100
170 A = 0:B = 347:C = 1:D =  - 8:E =
     8:K = E
177 :
178  REM *** MOVE SPRITE 0
179 :
180  FOR X = A TO B STEP K
190  IF Q = 0 THEN  POKE V + 23,1: POKE
     V + 29,0:Q = 1: GOTO 210
200  POKE V + 23,0: POKE V + 29,1:Q =
     0
207 :
```

```
208  REM *** DELAY MOVEMENT
209 :
210  FOR Z = 0 TO 250: NEXT
220 HB =  INT (X / 256):XP = X - 256 *
     HB
230  POKE V,XP: POKE V + 16,HB
240  NEXT
247 :
248  REM *** MOVE <- OR -> ?
249 :
250  IF A = 0 THEN A = 347:B = 0: POKE
     V + 27,B:K = D: GOTO 180
260  IF A = 347 THEN A = 0:B = 347: POKE
     V + 27,C:K = E: GOTO 180
397 :
398  REM *** DATA FOR SPRITE 0
399 :
400  DATA 1,0,0,1,0,0,7,128,0,7,224,0
     ,7,240,0,7,112,0,15,48
410  DATA 7,255,0,31,255,0,63,255,0,1
     11,254,0,207,254,0,207,251,0,6,1
     ,128
420  DATA 6,1,128,6,1,128,128,12,1,12
     8,24,1,128,48,0,0,0,0,0,0,0,0
```

Program 10.1.

Line-by-line analysis

Line 10: First we must set up the screen and border colours. They are, of course, purely a matter of choice, though that choice must take into account the colour, or colours, of the sprite(s) that you are using.

Line 20 tells the computer where to look for the DATA for sprite 0. The address for the POKE is found by adding the number of the sprite to the base address - 2040. The value to be POKEd is found by dividing the address of the first byte of DATA by 64. In this case the DATA will start at 832 - 832 divided by 64 equals 13. You may *not* start sprite DATA at any address which is not a multiple of 64.

Lines 30-50: A simple loop to collect the sprite DATA from lines 400-420 and POKE it into memory.

Line 100: Assigns the base address (the lowest byte of the Video chip) to the variable V.

Lines 110-160: Set up the basic parameters for sprite 0 in the following order:
(1) Clearing V+27 to 0 means that our sprite will appear in front of everything else on the screen.
(2) Setting the first byte of V+21 to 1 'enables' sprite 0 – enables us to see it, that is.
(3) The value POKEd to V+39 dictates the primary colour for sprite 0 using the standard colour values for the C64. The effect of this instruction *may* be altered if we also use the 'multi-colour' option (see below).

Because I wanted to alter the shape of my sprite as it moved, I've set the values of V+23 and V+29 so that the sprite will start off with normal width but double height. If the appropriate bits are not set (to 1) then a default value of 0 is used by the computer and the sprite will appear at normal width and height.

Finally I have POKEd the *horizontal* location for the sprite (that is the 100th graphics line from the top of the screen) into the relevant Y co-ordinate. The computer uses this as the location of the *top* row of bytes of the sprite.

Line 170: I'll explain this line in a little more detail in a moment. For the time being all we need to know is that, in the loop below, A is used as the location at which the sprite will appear and B is the furthest location (X co-ordinate) to which it will move on the far side of the screen. The value of C is used when we want the horse to appear in front of the background. D is the number of bytes that the sprite will 'jump' on each move when it is travelling backwards (right to left), and the value of E is used for the same purpose when it is travelling forwards (left to right). These two values are transferred to K as necessary. We could also assign a value to Q at this point, but this is unnecessary as yet, as unassigned variables are automatically cleared to 0 – which is the correct value for Q at the point when the routine starts.

Lines 180-240: This is a pretty complicated loop, used to move the sprite across the screen, so I'll explain it in detail.

Line 180: As I said before, the sprite actually jumps across the screen rather than moving one byte at a time. A and B are the start and finish location for each journey, and the value of K dictates how many bytes it will jump on each move. You might notice that despite this jump in the loop the sprite still appears to move quite smoothly.

Lines 190-200: In order to make the horse 'bounce' I have to continually reset the bytes controlling its width and height. Thus on every move the value of Q is changed to 1 or 0, depending on its previous value, and that dictates whether it is double height, normal width (line 190), or double width, normal height (line 200). Only *one* of these lines is executed on each move.

Line 210: This is a simple 'wait' loop to give us time to see the sprite before it moves again.

Lines 220: This line is essential to the process of getting the sprite all the way over to the right-hand edge of the screen because V+16 holds a second (high byte) value for the X co-ordinate, which allows us to get a total value of up to 347 - the furthest location on the right-hand side of the *graphics* screen.

Line 250-260: As with line 190-200 there is a selection system to decide which of the two lines will be executed, except that here the lines alternate with each *loop* and not on each move.

If A=0 when the program reaches this point then the horse has been moving from left to right, which means that we now want to send it back again. To do this we reverse the values of A and B, and K - the 'step' controller - must be given the value of D, a negative value, because the loop will be counting down, not up. But I also wanted to demonstrate another function of the VIC chip, the ability to move sprites *behind* any background material. To do this we POKE the new value of B (=0) into the priority register.

The elements of line 260 are simply the reverse of those in line 250, but they will only be used where the loop ends with the value of A set to 347. In this case we know, or rather the computer 'knows', that the sprite has completed a journey from screen right to screen left. So once again the values of A and B are swapped, K is set to equal E - a positive number - and the value of C (=1) is POKEd into V+27 to put the sprite back in front of anything else there may be on the screen. At the end of both of these lines program execution is sent back to the start of the loop, at line 180, for a new journey.

Just a couple of extra points. If you want to use a multi-coloured sprite then add this line to the program:

```
125 POKE V+28,1: POKE V+37,12: POKE V+38,13
```

POKEing the appropriate *bit* in V+28 tells VIC that you want sprite (bit no. −1) to be multi-coloured, using the values in V+37 and V+38. In this example the two secondary colours will actually

block out the original black used for this sprite altogether, though some combinations will give you a three-coloured sprite. Experiment with the last two POKEs in line 125 to see the different effects.

Lastly, in order to stop this program simply press the RUN/STOP key. If the sprite is still on the screen you can erase it by typeing in POKE 53269,0.

What we also need to know, of course, is how to prepare a sprite diagram so that we can compile the right set of DATA to include in our program. This is a fairly simple process, thanks to the way that Commodore have set up the sprite handling functions, and the only aspect that might present any problems at all is the job of labelling your 'sprite grid' correctly. In the two diagrams below (Figs 10.1 and 10.2) you will find first the basic sprite grid outline, and then a grid showing the rough diagram from which I coded the 'horse' sprite used in Program 10.1.

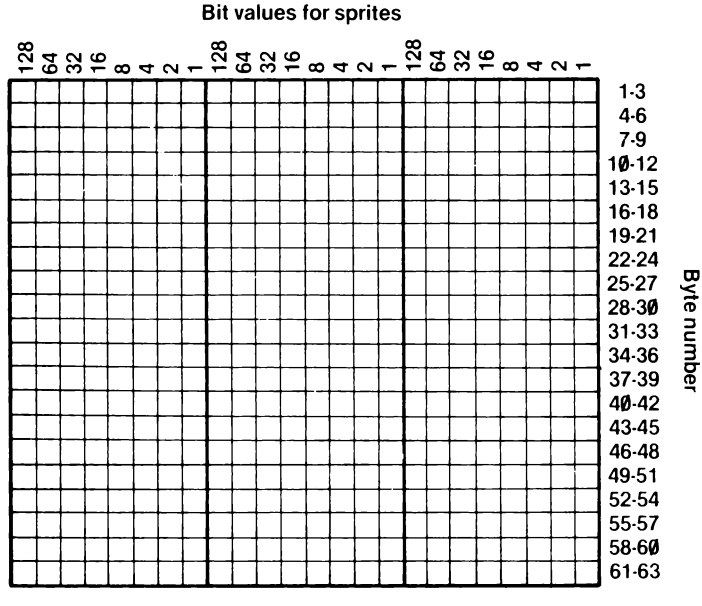

Fig. 10.1. Sprite grid.

You will see from Fig 10.2 that when you do prepare your sprite DATA each *byte* is taken in logical sequence moving across each row and then down to the next line, so that you *start* at the top left-hand corner and work your way down to the bottom right-hand corner. You may also notice that although each set of sprite DATA is assigned to a set of 64 locations, the data that is actually used by the computer is only 63 bytes long (that is, 3 columns by 21 rows).

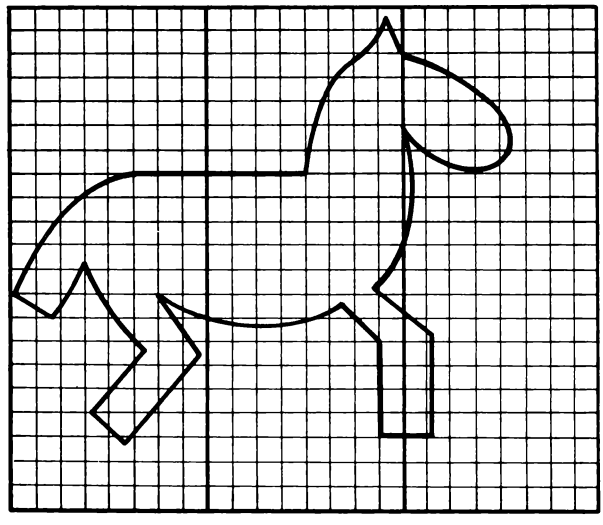

Fig. 10.2. Sketch of horse sprite.

The difference is simply because the computer finds it easier to calculate in multiples of 64 than in multiples of 63. The contents of the 64th byte are, therefore, totally unimportant and will not affect the sprite concerned.

Sounds good

Well now, if you think that sprite handling is a bit tricky just you wait till you start on the sound routines. With sprites you have only to draw and then code the sprite DATA correctly in order to get exactly what you expect. When using the sound routines, however, getting the right collection of notes (see the *User's Manual*, pages 152-154) is only the start of your problems. Once you know what notes you want to play you still have to define the sound 'envelope', choose a waveform, decide which voice (or voices) you want to use, etc. And then you have to make sure that all the relevant registers are POKEd in the right order!

The fact that sound routines *are* so difficult to access is really rather frustrating, because the actual hardware - the SID (Sound Interface Device) chip - is actually a very advanced device worthy of synthesiser units selling at several times the cost of the entire C64. Don't feel too bad, then, if you still find the task of accessing SID about as straightforward as a trip through Hampton Court maze. This chapter should at least give you a reliable map!

So, if it's that difficult to use the sound routines, why bother with them at all? In fact I think the *useful* inclusion of these routines is limited to three settings within any adventure:

(1) As part of the Introduction routine.
(2) To give short sound signals to indicate good and bad moves finding a useful item, falling into a pit, etc.
(3) A pair of End Of Game melodies – a winning tune and a losing or 'I give up' tune which use the same routine but different sets of DATA.

In other words, unless you manage to complete an adventure program and still find yourself with plenty of RAM to spare (a rather unlikely event), or you have moved on to the point where your sound and graphics routines are in machine code, stored in one of the 'double- mapped' areas of RAM, and can be activated with a simple SYS command, then the use of sound and vision should definitely be regarded as an expensive luxury rather than a necessity.

Having said that, since the sound facilities *are* so good I'd like to finish this chapter by exploring them in a little more detail. (For a far fuller explanation I would suggest you get hold of a copy of Steve Money's excellent new book *Commodore 64 Graphics and Sound*, also published by Granada).

What's where in SID

Since the *User's Manual* really doesn't give a very clear memory map of the sound chip (and every edition appears to include at least one incorrect address) let's start off with a list of the relevant locations, or *registers*, together with their functions.

And now, in case you're thinking that all that detail has made matters worse rather than better, let's put it all to some use. Once you get the hang of it, it isn't half as difficult as it may look.

The central feature of any SID-based operation is the need to 'enable' whichever voice (or voices) you wish to use. At the simplest level this involves POKEing just six pieces of information into the chip registers:

(a) The ATTACK/DECAY value
(b) The SUSTAIN/RELEASE value
(c) The WAVEFORM value

(*Note:* values (a) and (b) must always be entered before value (c).)

SOUND INTERFACE DEVICE (SID)

Base Address (=BA):54272
Top of SID: 54300
Size in BYTES: 28
Note: The first 24 locations in SID are WRITE ONLY - their true values *cannot* be PEEKed.

SID CONTENTS

Absolute Address	*Relative Address*	*Contents*
54272	BA+0	Low Frequency - Voice 1
54273	BA+1	High Frequency - Voice 1
54274	BA+2	Pulse Width (Low) - Voice 1 (0-255) **Note**: The Pulse registers are only used in conjunction with the Pulse Waveform - 65.)
54275	BA+3	Pulse Width (High) - Voice 1 (0-15 only)
54276	BA+4	Waveform - Voice 1 (enter 17,33,65 or 129 or) plus Sync./Mod. (POKE Address, 19 (17+2) for Sync., or POKE Address,21 (17+4) for Mod.)
54277	BA+5	Attack/Decay register - Voice 1
54278	BA+6	Sustain/Release register - Voice 1
54279	BA+7	Low Frequency - Voice 2
54280	BA+8	High Frequency - Voice 2
54281	BA+9	Pulse Width (Low) - Voice 2
54282	BA+10	Pulse Width (High) - Voice 2
54283	BA+11	Waveform - Voice 2 plus Sync./Mod.
54284	BA+12	Attack/Decay register - Voice 2
54285	BA+13	Sustain/Release register - Voice 2
54286	BA+14	Low Frequency - Voice 3
54287	BA+15	High Frequency - Voice 3
54288	BA+16	Pulse Width (Low) - Voice 3
54289	BA+17	Pulse Width (High) - Voice 3
54290	BA+18	Waveform - Voice 3 plus Sync./Mod.
54291	BA+19	Attack/Decay register - Voice 3
54292	BA+20	Sustain/Release register - Voice 3
54293	BA+21	Low Cutoff for filters (0-7 only)
54294	BA+22	High Cutoff (0-255)
54295	BA+23	Filter enable (set bit Voice number −1) plus Resonance - bits 4 to 7

Absolute Address	*Relative Address*	*Contents*
54296	BA+24	Volume (0-15) plus filters (set bit 4 for Low Pass, bit 5 for Band Pass and bit 6 for High Pass). When bit 7 is set (=1) Voice 3 is turned off.
The last four locations in SID are all READ/WRITE addresses and may be PEEKed.		
54297	BA+25	Holds value for Paddle 1 if in use
54298	BA+26	Holds value for Paddle 2 if in use
54299	BA+27	Keeps track of the value in 54290. It is then incremented at a rate which depends on the *frequency* of Voice 3
54300	BA+28	Works in a very similar manner to the last register but is controlled by the Voice 3 *envelope*.

Table 10.2. Structure of the SID (sound interface device).

(d) The VOLUME LEVEL for the sound
(e) The HIGH FREQUENCY value
(f) The LOW FREQUENCY value

Obviously there are several other bytes which *could* be adjusted as well for fuller note control, but these are the only ones that *must* be set up by the program. The only other value to be set by your program – as a loop rather than by POKEing it – is the note duration. Thus a simple program to set up Voice 1 to play a single note might look like this:

```
10 SC = 54272: K = 57
20 FOR X = 0 TO 24
30 POKE SC + X,0
40 NEXT
50 POKE SC + 24,15
60 POKE SC + 5,1
70 POKE SC + 6,240
80 POKE SC + 4,17
100 POKE SC,172: POKE SC + 1,K
120 FOR X = 1 TO 500
130 NEXT
150 FOR X = 0 TO 24
160 POKE SC + X,0
170 NEXT: END
```

The segments of the program break down as follows:

Line 10: SC represents the 'base address' for the sound chip. K is the High Frequency value for the note A.

Lines 20-40: Clear any values previously held in the first 25 registers.

Line 50: Sets the VOLUME register to its maximum value.

Line 60: Sets the ATTACK/DECAY register for Voice 1 to 1 - very fast attack and decay times.

Line 70: Sets the SUSTAIN/RELEASE value for Voice 1.

(**Note:** the values for the ATTACK/DECAY and SUSTAIN/RELEASE registers are calculated by assuming that each *element* may have a value in the range 0-15. The value to be POKEd to the register is found by the formulas AD=(AT*16)+DE or SR=(SU*16) + RE.)

Line 80: Sets the WAVEFORM for Voice 1 to 17 - Triangular Wave.

Line 100: Loads the first and second registers of the chip with the *low* and *high frequencies* for the note to be played. As soon as this line has been executed the note will begin, and will continue until SC and SC+1 are cleared to zero.

Lines 120-130: A simple FOR...NEXT loop keeps the note going for a fixed time until...

Lines 150-170: A repeat of lines 20-40 to clear all the first 25 registers. The note will cease (with a slight 'click') as soon as the loop is completed for the second time.

So far, so good. But how do we vary notes, or add another voice?

The answer to the first question is usually to add a set of DATA statements to the end of the routine containing the High and Low frequency values of the notes to be played plus, if you want to vary the length of the notes, a set of Duration values. If, then, the DATA is stored in lines 160-180 or whatever, lines 100-130 might look like this:

```
100 FOR X = 1 TO (total number of notes)
110 READ LO,HI,DU
120 POKE SC,LO: POKE SC + 1,HI
130 FOR Y = 1 TO DU: NEXT
140 NEXT
```

Alternatively, and as a simple way of testing out a range of notes (in two voices yet!) make the following amendments to the original program:

```
10 SC = 54272  (delete value for K)
60 POKE SC + 5,0: POKE SC + 12,30
70 POKE SC + 6,240: POKE SC + 13,100
80 POKE SC + 4,17: POKE SC + 11,33
90 FOR K = 20 TO 140 STEP 10
110 POKE SC + 8,140-K: POKE SC + 7,120
140 NEXT
```

And that's about it! From this point on everything becomes a matter of trial and error. Only a detailed knowledge of the technical aspects of sound generation can help to speed this process up.

Fortunately the C64 has already proved itself extremely popular and a number of sound generation routines – with and without graphical displays – have already begun to appear in various computer magazines, so be sure to watch out for them. To finish off this brief introduction here are a few short hints on some of the intricacies of the SID chip.

(a) As I said before, always set up the Attack/Decay and Sustain/Release registers for a voice *before* entering the Waveform. If you want to change AD or SR whilst a sound routine is running then re-set the Wave register afterwards.

(b) If you want to use the Sync. or Mod. facility for any voice then the second or third bits of the appropriate Waveform register must be set to 1. Note that these functions are normally only applicable if the Voice in question is set for Triangular wave (i.e. 17).

(c) *Very important!* All registers from 53272-54296 inclusive are *write only* addresses. If you try to PEEK them, directly or as part of an adjustment, you will always get a zero value returned. Thus the values for these registers, in any routine which involves adjustments such as turning filters on or off, etc., should always be held as variables or POKEd to a separate set of locations which can be copied to SID.

(d) In order to change the filters used in any routine you will need to set, or clear, bits 4 (Low Pass), 5 (Band Pass) or 6 (High Pass) of the Volume register at 54296. At the same time, however, bits 0-2 of 54295 must also be set (to filter a voice) or cleared (for unfiltered sound) – bit 0 relates to Voice 1, bit 1 to Voice 2, and bit 2 to Voice 3.

(e) By the same token, the Cutoff registers at 54293 and 54294 will only affect the sound SID is producing if the Filter registers have been set.

(f) The Pulse Width registers for each voice (54275 and 54276, etc.) will *only* affect the sound of a particular voice if it is set for the Pulse waveform (= 65).

Chapter Eleven
What Now?

In this last chapter I want to look at some of the exciting developments that will certainly occur in the area of adventure gaming – and computing in general – over the next few years; but first I have one more piece of programming for you. Writing adventure games can be every bit as satisfying as playing them – as I hope I've shown you already. But it's still fun to play them! If you'd like to turn to the Appendix at the end of this book you'll find the *complete* listing for a game I've called *The Case of the Missing Adventure*. I hope you enjoy playing it as much as I enjoyed writing it.

When I first began work on this book my primary intention was to provide a *useful* guide to the art of writing adventure games – the sort of book that would allow almost anyone with a computer and a reasonable knowledge of BASIC to create their own adventures, for fun or profit, without feeling they had to be an 'expert'. In fact, I have stuck entirely to BASIC, as I felt this was the best way to show readers *how* the various sections of an adventure game function.

Speeding things up

In practice, of course, a growing number of commercial games are written in machine code. Now using machine code doesn't often allow you to do anything that you couldn't do in BASIC. It does, however, allow the computer to execute the more complicated routines at much higher speed. The room description decompiler is one very good example of a routine which would be much improved by conversion to machine code.

As for the more advanced command parsing routines, the more sophisticated they become the more impractical it is to use them in a BASIC program because of the time taken to interpret just one

command. Indeed even in *Valhalla*, which has an excellent command routine written in machine code, the time taken to handle a single command begins to get quite noticeable after about half an hour of play.

But don't worry too much if you don't feel up to using machine code for while. The first adventures were actually written in FORTRAN, and even now quite a few professional games are first *written* in a form of BASIC – on a mainframe computer. The original program is then compiled (converted to machine code *by the computer*!) and the result is translated again into the particular form of machine code (6502, Z80, etc) used by the target micro.

On the other hand, those of you who are really into programming and are learning assembly language and machine code might like to add a practical dimension to your studies by converting some of the BASIC routines in this book into low level language.

Two heads ...

There are one or two best-selling adventure writers who prefer to work on their own – or at least started out that way. It's much more common, however, to find at least two writers, or even a whole group working on each project – Woods and Crowther, Blanc and Lebling, the Legend group (responsible for *Valhalla)*, and the Melbourne House group. My personal feeling is that the group setup is probably best, especially for newcomers, for several reasons:

(1) It's usually easier to generate fresh ideas in a group where ideas can be tossed back and forth.
(2) Working with at least one other person tends to create a more disciplined atmosphere. It's all too easy, when working alone, to get side-tracked by minor considerations. This is less likely to happen when there's someone around to ask what you're up to if you go off course.
(3) The group setting is ideal for on-the-spot appraisals of a program while it is still coming together. The lone writer may well try to get the whole task completed before going back to iron out the problems. This may produce the same end result, but it is likely to take much more time than a group effort, where each step of the process is being discussed and reviewed.
(4) Groups cater for specialised attention to specialised tasks.

No one factor on this list is more important than any of the others.

But the last factor is worth considering as a guide to the variety of tasks involved in creating an adventure game, and the numerous skills required. In fact there are seven different job descriptions, though two relate only to adventures with graphics.

(a) The 'ideas man' – has the task of coming up with as many ideas, detailed and general, as possible. It is his or her job to organise and lead creative discussions and 'brainstorming' sessions in the group when needed.
(b) The writer – is responsible for turning ideas into coherent story-lines. This person also prepares text for screen displays (the introduction and room descriptions) and for the documentation for each game.
(c) The map maker - takes the work of the ideas man and the writer and prepares a detailed map for each adventure including objects, booby-traps, etc. The 'creative team' – the ideas man, the writer and the map maker – are jointly responsible for the accuracy of the map and for providing full lists of items, characters, events and so on for use by other members of the group.
(d) The graphics designer – roughs out *all* screen displays, both those involving 'pictorial' presentation and purely textual displays like the character status report, inventory listing, etc. In a commercial operation he would also be required to design the packaging for the finished game with its documentation.
(e) The graphics programmer – is gradually becoming an essential part of adventure writing groups. Each manufacturer has a quite unique system for producing screen graphics, even though their particular version of BASIC is quite standard. When the graphics in a game are of a good to excellent standard it's often because it took as much time to write the graphics routines as it did to program the rest of the game. (Which is another reason why the majority of adventure games tend to use few if any graphic effects.)
(f) The analyst – covers two tasks found in commercial computing, the Consultant and the Systems Analyst. The adventure group analyst is involved in nearly all aspects of the preparation of each game. He or she must prepare a general outline of the game when it is first developed by the creative team and must monitor it throughout its development to ensure that it remains credible in terms of the amount of RAM space needed/available and to ensure that the ideas being put forward really are possible in pure programming terms. Once all the details of a game are complete the analyst must draw up a detailed flow chart for the programmer and ensure that

the documentation supplied by other members of the group is as clear and detailed as it needs to be.

(g) The programmer – has the thankless task of turning everyone else's ideas and efforts into a working program. In the case of a graphics adventure the programmer and the graphics programmer will, of course, work together as far as possible (though much of the preliminary coding will have to be done separately).

(h) The translator – takes games created on one machine and modifies them for use on different models. If you're thinking of setting up any kind of commercial operation then the ability to present the same game for a variety of machines is essential. If the game is successful and you don't market it for other machines then you can bet your life that someone else will. Just look what happened to *Donkey Kong* and *Frogger*!

The method I've outlined here may seem to be particularly concerned with the production of programs for the commercial market. Not so. What I've tried to do is to split up the creation of almost any game into its logical parts. Obviously you don't have to have a separate person for each task, though as computing becomes ever more popular I suspect that groups of about this size already exist all over the country. Not, perhaps, for the purpose of producing games, but rather because the members share common, computer-related interests.

Moreover, groups such as the one I've described need not be restricted to competent computer users. Several tasks can be undertaken by people who may have no interest in computing whatever apart from playing games on them. Indeed, it would be useful to have at least one person in each group who *doesn't* know much about computing: he can act as the 'guinea pig' who deliberately tries to crash the program when it's thought to be complete. (There wouldn't be half as many 'dud' programs on the market if some of the commercial games producers bothered to test run their programs before releasing them.)

But why put so much organisation, time and effort into creating games if you *don't* want to write commercial games? The answer might be 'Adventure Networks': large and small groups of people, all over the country, prepared to exchange ideas, tips, techniques and even complete adventure games on a one-for-one basis (don't forget to make copies of each game).

After a while some of the most productive groups might choose to 'go professional' – but there will always be another generation coming along to take their place. After all, that's almost exactly how the very first computer adventure game got started!

Why stick at games?

At last there really is a way of making learning more of a pleasure than a headache! Like most writers on the subject of computer adventures I've concentrated on the idea of adventures as games, that is, as leisure-time amusements. But the potential use of games covers a much wider area than pure entertainment.

One of the most obvious uses of adventure games, and one that is still almost entirely overlooked, is in the field of educational software. If adventure games are largely about solving problems – with some kind of reward for each *correct* solution (like being allowed to stay alive!) – then why not base adventures on specific subjects.

Biology, Chemistry and Physics are perhaps the most obvious place to start, with History and Geography close behind. The method of application will, I think, be fairly obvious. Chemistry is perfect for some kind of detective/mystery story, while Physics may be better suited to an all-action game. It would probably be wise to stick to just one subject per adventure, but the range of possibilities is endless. Indeed, since students are expected to have set standards of knowledge of their subjects you can aim at very clearly defined groups in terms of age and exams to be taken.

The educational software available at the time of writing tends to be either very good or, more often, pretty awful. The problem is that the market for this kind of program is so large, and has such a large potential for fat profits, that it is still a happy hunting ground for 'cowboy' programmers. As school staff become more familiar with computers, and computer programs, the demand for top rate software is bound to clean out the rubbish. But the market itself is likely to go on growing for some time yet. So there's still plenty of room for well-produced, relevant programs, especially those produced by students and teachers, who are ideally situated to know what is required.

You ain't seen nothing yet

So where does computer adventuring go from here? Certainly I don't know any more than you about what Uncle Clive and the rest of the computer manufacturers have in store for us next. All the same, here are a few guesses (I hesitate to call them predictions) about what will be coming our way in the next two to five years.

Large Memory. Believe it or not, even the APPLE only had 4K of RAM when it first appeared in 1977. Even in the last couple of years RAM space on the average micro has doubled and even trebled, and there is no reason to believe that the same kind of expansion of the basic memory won't continue, especially when the next generation of chips arrives and forces prices down. I will be most surprised if 128K micros don't become standard within the time period I've quoted, and this is going to make a lot of difference to the adventure programs of the future.

Tape to Disk. It has been common in the past, and is still true to a certain extent, for a single disk drive to be more expensive than the computer that it serves. However, prices are beginning to drop significantly, and it is highly likely that those people who have had a computer for two or three years are beginning to think very seriously about moving on from the old cassette. Hardly a surprising development when you think that one 5-inch floppy carries at least twice the storage space of a micro and takes only seconds to write to or read from.

8, 16 and 32 bits. Most readers will probably already know that the 'bit' size of a micro relates to the maximum number that the computer can handle in a single operation. The much-predicted move from 8-bit machines to 16-bit machines for the home market has already arrived, with the Sinclair QL. It only remains to see how quickly the other manufacturers can follow Uncle Clive's lead.

Micro Networks. The ability of numerous home computers to access a central database is almost taken for granted in America, where The Source is probably the best-known network to date. The use of such networks depends very largely on the wide distribution of home modem units (which connect computer to phone to phone to computer). Although modems are not exactly best-selling items in Britain at the moment, I think it's pretty likely that they will become so in about three to four years time. In this connection (sorry!) it's worth noting that modem interface cards are themselves becoming more sophisticated, and it's not impossible that you will eventually get a CPM-type interface where you can connect your micro to any other, regardless of make.

Now, how does all this relate to adventure games?

First there's the matter of storage capacity. The larger the standard RAM space becomes, the more detailed and complex adventure plots can become. Add to this the spread of disk drive systems rather than cassette-based systems and we could be talking about some very big games indeed! Moreover, when we combine

extended memory with faster chips (both 8- and 16-bit), not to mention the development of highly specialised graphics and sound chips, it's not difficult to imagine that graphics adventures (in the true sense), including the element of animation as seen in *Valhalla*, will become much more commonplace. Whether they will actually take over from text games, however, is something else again, and is likely to depend on the quality of the games themselves.

Some readers will already have seen the latest generation of arcade games which feature genuine cartoon action using a laser disk (!) in much the same way that the normal computer accesses a floppy or hard disk. Certainly plans are being made to produce laser disks for use with home micros, but just how soon this will actually come about is anybody's guess at the moment.

So I said 'Get the axe ...'

As I explained in Chapter 8, handling complicated command input requires a lot of space – and time. For the time being, the vocabulary and input language of adventure games is likely to go on improving, but I anticipate that in the long run, and as the hardware gets cheaper, adventure writers will become more and more interested in dealing with direct *verbal* commands from the player. (If you have to use that much space anyway, why not use it to the best advantage!) It shouldn't be too long, then, before players can save their fingers by feeding commands directly to the computer via a microphone – and the computer will, of course, answer back!

So, more pictures, animated action, bigger games and spoken communication. What else could there be? The something else may well be, I would imagine, the spread of adventure networks. Not the sort of network that I described earlier in this chapter but groups where several people can all take part in a single adventure without going outside their own front door!

By way of their modem connections I can imagine adventures becoming part of nationwide link-ups where each player can join in, or bow out, of an ongoing game. The forerunner to this idea – postal adventures monitored by a central computer – has already proved to be a commercial success on a limited scale. And the idea of direct 'home-to-mainframe' phone-linked adventuring is now in use on a minor scale at Essex University. Needing only a terminal, each player can sign on in a game – known as MUD (Multi-User Dungeon) – in progress on a PDP-11 computer.

In a sense, if and when this does happen on a large scale, adventuring will have gone full circle – from group activity to solo game to group activity. Yet in completing that circle it will have evolved and developed in a way Gygax and Arneson could hardly have imagined when they first broke away from their own boardgame group in order to develop *Dungeons and Dragons*.

It will, I hope, prove to have been a positive development.

Appendix

The Case of the Missing Adventure

This fascinating case, the last *known* exploit of the famous computer detective – Mike Rowchip – involves the search for an adventure game lost somewhere in the vast building occupied by Fantasia International, a highly successful software publishing house.

Believing that the game has been stolen by a rival company, Mike goes in search of Eddy the Kwill, an underworld character whose willingness to tell what he knows (for a price!) has made him less than popular with other members of the criminal fraternity. After several days following Eddie's devious trail, Mike finds himself at a dead end. What he doesn't know is that this particular dead end lies at the back of the 'new projects' section of the Fantasia International building ...

Fortunately the projects section only takes up ten rooms in the F.I. building, and the missing adventure is definitely in one of those rooms. Your task is simply to find it. If you are successful then, under the Finders/Keepers law introduced in 1987, it becomes your property. Good hunting!

So that's how it's done ...

Unlike the rest of the programs in this book I have *not* included a line-by-line analysis for the game (no, you haven't bought a copy that has the last few pages missing!).

By the time you reach this point in the book you will, I hope, at least have thumbed through the various programming modules with their explanations. Broadly speaking there is nothing in the game program that is not dealt with, in detail, elsewhere in the book. Having said that, I would like to mention just a few minor points that may be of help to anyone who wants to break this program down to see exactly how it works.

(1) *The Structure.* As far as possible I have used modules exactly as they appear in other parts of the book – with altered line numbers, of course. At the same time I tried to avoid making the general flow of the game too obvious so that readers won't learn too much about the game simply by entering it – there isn't much point in playing the game if you already know the solution! So, the various routines within the game have been laid out in a fairly random fashion. But, each routine is entirely self-contained except in about three cases where a single routine does two or more jobs. Moreover each module starts on what might be termed an 'even numbered multiple of 100' – 7000, 10800, 11200, etc.

(2) *Routine variations.* To repeat what I said before, most of the modules within the game are in the same form that they appear elsewhere in the book – with a couple of variations. Firstly, the flow *from* the Command Parser to the rest of the program is not controlled by the formula X=V*NT+N–NT that I used in Chapter 8. Instead each *verb* has its own area of the program and this is reached by the simplified command ON (verb number) GOTO.

You will also find a couple of extra subroutines which don't appear elsewhere in the book. I won't tell you much about these modules except that the most important one will only become obvious when you try to map out the game plan. There is, by the way, a clue in the game itself as to what this subroutine is doing.

(3) *Extra keys.* In the last section of the program you will find two extra bracketed commands – CTRL 1 and CTRL 6. These are used simply to change the colour of the lettering and produce White text and Green text respectively. To use them hold the Control key and the Number key down *at the same time.*

You will also notice that I have used the '2 Letter' command parsing routine. In order to use it please consult the list of verbs and nouns given below. Incidentally, this list is complete. There are *no* hidden commands.

All of the verbs should be accompanied by a 'logical' noun, except for verbs A and B, or the command will be rejected.

(4) *Line Length.* You will find that this listing includes a few lines with numbers ending in 5, where 99.9% of the lines end in zero. The reason for this lies in the two-line limit for entering statements on the C64. In fact the program *will* RUN as listed, but if you use the abbreviations listed in the User's Handbook, you will find that the contents of the odd-numbered lines will fit onto the end of the previous line.

I mention this in order to introduce one last tip. If you have ever

Verbs	*Nouns*
A - Inventory (Note: No second letter is required with verbs A and B)	A - not used
B - Search (as in LOOK or EXAMINE)	B - Flowchart
C - Go	C - Map
D - Get	D - Storyline
E - Drop	E - Key
F - Use	F - Hatchet
G - Read	G - Crucifix
H - Open	H - Badge
I - Unlock	I - Ring
J - Climb	J - Ray Gun
K - Chop Down	K - Stairs
L - Kill	L - Tree
	M - Safe
	N - Vampire
	O - Gobbet
	P - Sign
	Q - Door
	R - North
	S - South
	T - East
	U - West
	V - Up
	W - Down

tried to 'pack' a line on the C64 you will notice that, unless the cursor is still on the second line, when you press <<RETURN>> you get a ?SYNTAX ERROR message, even though your line is within the 80 character limit. If you ever need to get round this there is a simple solution: enter your statement or text up to the last place on the second line (so that the cursor is on the first position of the next line) and then press <<RETURN>>. Ignore the error message and, without LISTing the line, use the cursor keys to get up to any position on the apparently rejected line and press <<RETURN>>. You should find that the line is now accepted in full and no further error message is given.

Back word

Just before we come to the game listing there are a couple of points that I'd like to make clear in case anyone has a problem with any of the programs in this book.

First, as I stated in the Note to the Reader at the start of the book, all the programs have been tested before publication. Unfortunately, as all programmers know from bitter experience, it is impossible to test every routine 'to destruction' and it is possible that a few bugs still remain undetected. If you should be unlucky enough to find one then please accept my sincere apologies.

Secondly, if you think you have found a 'bug' which you are unable to remedy, or if you have any other queries about this book, then please *don't* contact Granada directly; they are publishers, not computer experts. If, on the other hand, you would like to write to me – care of Granada – enclosing a stamped addressed envelope, I will do my best to deal with your enquiry as quickly as possible.

And now for *The Case of The Missing Adventure* ...

```
1 REM *****  THE CASE OF THE MISSING
2 REM *****          ADVENTURE
3 :
4 REM *****      BY A.J. BRADBURY
5 :
6 REM *****       COPYRIGHT 1984
9 :
10 :
1000 GOSUB16000
1100 AD=INT(RND(1)*9)+1
1500 GOTO5000
2000 PRINT"YOU ARE CARRYING: "
2010 FORX=1TO10:IFOB$(X,1)="-1"THENPRINT"A
"OB$(X,0)
2020 NEXT
2030 PRINT"A TOTAL WEIGHT OF "IN" LBS."
2040 GOTO7000
2200 IFSF<>3ORPL<>ADTHEN2250
2210 PRINT"[DOWN * 3]"TAB(9)"[RVS]CONGRATUL
ATIONS!!![OFF]"
2215 PRINT"[DOWN] YOU'VE FOUND THE MISSING
ADVENTURE"
2220 PRINT" NOW ALL YOU NEED IS A GOOD SOFT
WARE    HOUSE TO MARKET IT AND MAKE ";
```

```
2230 PRINT"YOUR FORTUNEFOR YOU.":PRINT" BUT
 THAT'S ANOTHER GAME ALTOGETHER ...":END
2250 IFPL=1THEN2300
2260 FORX=2TO4
2270 IFVAL(OB$(X,1))=PLTHENPRINT"YOU FOUND
"OB$(X,0)"!":O1=1
2280 NEXT:IFO1=1THENO1=0:GOTO7000
2290 GOTO2320
2300 PRINT"THE OBJECT IS A SMALL BADGE WITH
 A BLACKFIGURE 6 ON IT.":GOTO7000
2320 PRINT"[DOWN]WHAT YOU SEE IS WHAT THERE
 IS!":GOTO7000
3000 PRINT"[DOWN]THE SAFE HAS A COMBINATION
 LOCK - ";
3005 PRINT"WHERE HAVE YOU HEARD THAT BEFORE
?."
3010 PRINT"WOULD YOU LIKE TO TRY TO OPEN IT
?"
3020 INPUT"(ENTER Y OR N) ";AN$
3030 IFAN$<>"Y"THEN7000
3040 LO=INT(RND(1)*999)+1:IFLO<100THENLO=LO
+100
3050 LO$=STR$(LO):LO$=RIGHT$(LO$,3):Q=0
3060 FORX=1TO5
3070 PRINT"PLEASE ENTER [RVS]DIGIT[OFF] NO.
"X;:INPUT"NOW ";SA$
3080 IFSA$<>MID$(LO$,X,1)THENPRINT"[DOWN]WR
ONG!":GOTO3100
3090 PRINT"[DOWN]CORRECT!!":Q=Q+1:IFQ=3THEN
3150
3100 NEXTX
3110 PRINT"[DOWN]YOU'VE GUESSED IT - THE SA
FE IS BOOBY-  TRAPPED.  YOU FALL THROUGH";
3120 PRINT" A HATCHWAY  INTO THE ROOM BELO
W.";:IFOB$(7,1)="-1"THENBT=1
3130 IFBT=1THENPRINT"  AND YOU'VE DROPPEDTH
E CRUCIFIX!"
3140 GOTO11850
3150 PRINT"[DOWN]OH DEAR! ... ":GOTO3110
4600 HF=0:PRINT"[DOWN]YOU CAN SEE ";
4610 FORX=5TO10
4620 IFVAL(OB$(X,1))=PLTHENHF=1:PRINT"A "OB
$(X,0)
4630 NEXT
4640 IFHF=1THENS1$="AND"
4650 IFHF=0AND(PL=1ORPL=3ORPL=7)THENS1$="NO
THING EXCEPT":HF=1:PRINT
```

```
4670 IFPL=1ORPL=3THENPRINT"[DOWN]"S1$" A SI
GN ON THE WALL."
4680 IFPL=7THENPRINT"[DOWN]"S1$" A SIGN ON
THE DOOR LEADING WEST."
4690 IFHF=0THENPRINT"ONLY WHAT IS LISTED AB
OVE."
4700 HF=0:RETURN
5000 FORX=1TO4
5010 T=INT(RND(1)*7)+3
5020 FORY=1TOX-1:IFVAL(OB$(Y,1))=TTHENY=X-1
:NEXTY:GOTO5010
5030 NEXTY
5040 OB$(X,1)=STR$(T)
5050 NEXTX
6000 PRINT"[DOWN * 5]":SH=12:PL=1:PRINTRD$(
PL):SF=0:FF=0
6010 GOSUB4600
7000 GOSUB9000:PRINT"[DOWN]WHAT NOW? ";
7010 GETV$:IFV$=""THEN7010
7020 FORX=1TO12
7030 IFV$=MID$(VE$,X,1)THENVP=X:X=12:NEXT:G
OTO7050
7040 NEXT:PRINT" ":GOTO7200
7050 PRINTV$(VP);
7060 IFVP=1ORVP=2THENPRINT"[DOWN]":PRINT"[R
VS]"V$(VP)":[OFF][DOWN]":ONVPGOTO2000,2200
7080 GETN$:IFN$=""THEN7080
7090 IFN$=BA$THENFORX=1TOLEN(V$(VP)):PRINTB
A$;:NEXT:GOTO7010
7100 FORX=1TO23
7110 IFN$=MID$(NO$,X,1)THENNP=X:X=23:NEXT:G
OTO7130
7120 NEXT:PRINT" ":GOTO7210
7130 IFNP<18THENPRINT" THE";
7140 PRINT" "N$(NP);:T=TI:T=T+120
7150 GETAL$:IFAL$=""ANDTI<TTHEN7150
7160 IFAL$=CHR$(20)THENFORX=1TOLEN(N$(NP))+
1:PRINTCHR$(20);:NEXT:GOTO7080
7170 PRINT" ":IFVP>6THENVP=VP-6:GOTO7190
7180 ONVPGOTO0,0,10000,10200,10400,10600
7190 ONVPGOTO10800,11000,11200,11400,11600,
11800
7200 PRINT"[DOWN]I DON'T UNDERSTAND [RVS]"V
$:GOTO7000
7210 PRINT"[DOWN]I DON'T UNDERSTAND [RVS]"V
$(VP)" "N$:GOTO7000
```

```
9000 ND=INT(RND(1)*4)+2:LR=6
9010 MS=VAL(RIGHT$(MC$,1))
9020 FORX=NDTOLR:MC(3,X)=VAL(MID$(MC$,X-1,1
)):NEXTX
9030 IFND<>2THENLR=ND-1:ND=2:GOTO9020
9040 MC(3,1)=MS:MC$="":FORX=1TO6:MC$=MC$+CH
R$(MC(3,X)+48):NEXTX
9100 IFOB$(1,1)="0"THENRETURN
9110 BL=VAL(OB$(1,1)):BM=INT(RND(1)*6)+1:IF
MC(BL,BM)=0THENRETURN
9120 BL=MC(BL,BM):OB$(1,1)=CHR$(BL+48):IFBL
<>PLTHENRETURN
9200 PRINT"[DOWN]OOPS!  YOU'VE JUST FOUND T
HE DREADED BUG- YOU HAVE NO ";
9205 PRINT"CHOICE BUT TO";
9210 PRINT" STAND AND   FIGHT ...":FORX=1TO
3000:NEXT
9220 ML=2:MH=21:SL=1:IFFF=1THENSL=9
9230 PS=INT((SL*(RND(1)*6+1)+SH)/6)
9240 MH=MH-PS:IFMH<1THEN9300
9250 MS=INT((ML*(RND(1)*6+1)+MH)/6)
9260 SH=SH-MS:IFSH<1THEN9400
9270 GOTO9230
9300 PRINT"[DOWN]WELL DONE - YOU'VE DEFEATE
D THE DREADED BUG.  YOU ARE ";
9305 PRINT"FREE TO CONTINUE."
9310 OB$(1,1)="0":RETURN
9400 PRINT"[DOWN]BAD LUCK - THE BUG GOT YOU
.  T-T-THAT'S ALL FOR NOW FOLKS!":END
10000 NR=NP-17:IFNR<1ORNR>6THENN$=N$(NP):GO
TO7210
10010 IFMC(PL,NR)=0THENKF$=" ":GOTO10050
10020 IFPL=7AND(NM=4ANDKF=0)THENKF$="- YET!
":GOTO10050
10040 PL=MC(PL,NR):PRINT"[DOWN]O.K.":PRINT"
[DOWN]"RD$(PL):GOSUB4600:GOTO7000
10050 PRINT"[DOWN]SORRY - YOU CAN'T GO THAT
 WAY "KF$:GOTO7000
10200 IFNP>10THENN$=N$(NP):GOTO7210
10210 FORX=1TO10
10220 IFN$(NP)=OB$(X,0)THENCH=X:X=10:NEXT:G
OTO10250
10230 NEXT
10240 PRINT"[DOWN]I DON'T SEE "N$(NP)" HERE
.":GOTO7000
10250 IFIN+VAL(OB$(CH,2))>20THENOF=99
```

```
10260 IFOF=99THENPRINT"[DOWN]SORRY - YOU CA
N'T CARRY ANYTHING THAT BIG.":OF=0:GOTO7000
10270 IFVAL(OB$(CH,1))=-1THENPRINT"[DOWN]YO
U ALREADY HAVE THE "N$(NP)"!":GOTO7000
10280 IFVAL(OB$(CH,1))=0THENPRINT"[DOWN]SOR
RY - THE "N$(NP)" ISN'T AVAILABLE!":GOTO700
0
10290 IFVAL(OB$(CH,1))<>PLTHENPRINT"[DOWN]T
HE "N$(NP)" ISN'T HERE!":GOTO7000
10300 PRINT"[DOWN]O.K. - YOU HAVE THE "N$(N
P)"."
10310 OB$(CH,1)="-1":IN=IN+VAL(OB$(CH,2)):I
FCH=3ORCH=4ORCH=5THENSF=SF+1
10320 IFCH=2THENFF=1
10330 IFCH=5THENMC(7,4)=10
10340 GOTO7000
10400 FORX=1TO10
10410 IFOB$(X,0)=N$(NP)THENTE=X:X=10:NEXT:G
OTO10440
10420 NEXT
10430 PRINT"[DOWN]SORRY - THERE IS NO "N$(N
P):GOTO7000
10440 Q=0
10450 IFVAL(OB$(TE,1))<>-1THENQ=1
10460 IFQ=1THENQ=0:PRINT"[DOWN]YOU CAN'T DR
OP WHAT YOU DON'T HAVE!!":GOTO7000
10470 PRINT"[DOWN]O.K.":OB$(TE,1)=STR$(PL):
IFTE=3ORTE=4ORTE=5THENSF=SF-1
10480 IFTE=2THENFF=0
10490 IFTE=5THENMC(7,4)=0
10500 IN=IN-VAL(OB$(TE,2)):GOTO7000
10600 IFPL<>5ANDPL<>5ANDPL<>7THEN10700
10610 IFPL=5ANDN$(NP)="HATCHET"THEN11610
10620 IFPL=4ANDN$(NP)="KEY"THENPRINT"[DOWN]
THE KEY DOESN'T FIT THE SAFE.":GOTO3000
10630 IFPL=7ANDN$(NP)="RING"THENPRINT"[DOWN
]AS YOU HOLD UP THE RING BULBOUS ";:R7=1
10640 PRINT"SIMPLY  FADES AWAY.  YOU ARE FR
EE TO CONTINUE.":R7=0:GOTO7000
10700 PRINT"[DOWN]USING THE "N$(NP)" HAS NO
 EFFECT.":GOTO7000
10800 IFPL<>1ORPL<>3ORPL<>7ORN$(NP)<>"MAP"T
HENPRINT"[DOWN]READ WHAT?":GOTO7000
10810 IFN$(NP)="MAP"THENPRINT"[DOWN]HARD LU
CK - IT ISN'T FOR THIS":PRINT"ADVENTURE.":G
OTO7000
```

```
10820 IFPL=1THENS$="[DOWN]DO YOU WANT INFOR
MATION?  WELL YOU WONT GET IT!"
10830 IFPL=3THENS$="[DOWN]THE WORLD KEEPS T
URNING - AND SO DO I.   "
10840 IFPL=7THENS$="[DOWN]ABANDON HOPE ALL
YE WHO ENTER HERE!!!    "
10850 PRINT"[DOWN]THE SIGN SAYS:":PRINTS$:G
OTO7000
11000 IFPL<>4THENPRINT"[DOWN]HUH?":GOTO7000
11010 IFPL=4ANDN$(NP)="SAFE"THENN$(NP)="KEY
":GOTO10620
11020 N$=N$(NP):GOTO7210
11200 IFPL<>7THENPRINT"[DOWN]NOTHING HERE I
S LOCKED.":GOTO7000
11210 IFN$(NP)="KEY"THENPRINT"[DOWN]O.K.  B
UT DON'T SAY YOU WEREN'T WARNED!!":K7=1
11220 IFK7=1THENNP=20:GOTO10000
11230 GOTO7000
11400 IFPL<>3ANDPL<>5ANDPL<>8THENPRINT"[DOW
N]THERE'S NOTHING HERE TO BE CLIMBED.":GOTO
7000
11410 IF(PL=3ORPL=8)ANDN$(NP)="STAIRS"THENN
P=22:GOTO10000
11420 IFPL=5ANDN$(NP)="TREE"THENPRINT"[DOWN
]WHOOPS - A BRANCH BROKE!":BB=1
11430 IFBB=1THENPRINT"AND YOU'VE BROKEN ONE
 ARM!":SH=SH-3:GOTO11900
11440 PRINT"[DOWN]YOU CAN'T DO THAT.":GOTO7
000
11600 IFPL<>5THENPRINT"[DOWN]THERE IS NOTHI
NG HERE TO CHOP DOWN.":GOTO7000
11610 PRINT"[DOWN]O.K.  YOU NOW HAVE A PILE
 OF LOGS.":GOTO7000
11800 IFPL<>7ANDPL<>8THENPRINT"[DOWN]THERE'
S NOTHING HERE TO KILL!":GOTO7000
11810 IFPL=7THENPRINT"[DOWN]BULBOUS IS UNDE
R A SPELL AND CAN'T BE   KILLED!";:F7=1
11820 IFF7=1THENPRINT"  BUT YOU'VE WASTED S
TRENGTH IN  THE EFFORT.":SH=SH-2:GOTO11900
11830 IFPL=8ANDOB$(7,1)="-1"THENPRINT"[DOWN
]THE VAMPIRE DISAPPEARS AT THE SIGHT ";:F8=
1
11840 IFF8=1THENPRINT"OF  THE CRUCIFIX. YOU
 ARE FREE TO CONTINUE.":GOTO7000
11850 PRINT"[DOWN]WITHOUT THE CRUCIFIX YOU
ARE POWERLESS  AGAINST THE VAMPIRE.":SH=0
```

```
11900 IFSH<1THENPRINT"OH DEAR - YOU'VE USED
 UP ALL YOUR        STRENGTH!  R.I.P.":END
11910 GOTO7000
13000 DIMV$(12),N$(22),MC(10,6),OB$(10,2),R
D$(10)
13010 FORX=1TO12:READV$(X):NEXTX
13020 FORX=1TO23:READN$(X):NEXTX
13030 FORX=1TO10:FORY=1TO6:READMC(X,Y):NEXT
Y:NEXTX
13040 FORX=1TO10:FORY=0TO2:READOB$(X,Y):NEX
TY:NEXTX
13050 FORX=1TO10:READA$,B$,C$,D$:RD$(X)=A$+
" "+B$+" "+C$+" "+D$:NEXTX
13070 VE$="ABCDEFGHIJKL":NO$="ABCDEFGHIJKLM
NOPQRSTUVW"
13080 BA$=CHR$(20):MC$="624598"
13100 RETURN
14000 DATAINVENTORY,SEARCH,GO,GET,DROP,USE,
READ,OPEN,UNLOCK
14010 DATACLIMB,CHOP DOWN,KILL
14100 DATATHE BUG,FLOWCHART,MAP,STORYLINE,K
EY
14110 DATAHATCHET,CRUCIFIX,BADGE,RING,RAY G
UN
14120 DATASTAIRS,TREE,SAFE,VAMPIRE,GOBBET,S
IGN,DOOR
14130 DATANORTH,SOUTH,EAST,WEST,UP,DOWN
14200 DATA2,0,0,0,0,0,3,1,0,0,0,0,6,2,4,5,9
,8,0,0,3,0,0,0
14210 DATA0,5,7,3,0,0,7,3,0,0,0,0,0,6,5,0,0
,0,0,0,0,0,0,3,0
14220 DATA0,0,0,0,0,3,0,0,7,0,0,0
14300 DATATHE BUG,3,0,FLOWCHART,3,2,MAP,3,2
,STORYLINE,3,2
14310 DATAKEY,7,1,HATCHET,6,16,CRUCIFIX,4,4
14320 DATABADGE,1,1,RING,5,1,RAY GUN,2,6
14400 DATAYOU ARE INSIDE FANTASIA INTERNATI
ONAL!  THIS FIRST ROOM IS
14410 DATAABOUT 10 FEET BY 6   WITH ONE WIN
DOW HIGH UP ON THE WEST WALL
14420 DATATHERE IS A SMALL OBJECT ON THE FL
OOR   AND WHAT LOOKS LIKE
14430 DATAA HANDLE ON THE FAR WALL.
14500 DATAWELL 'BEAM ME UP SNOTTY' - YOU'RE
 ABOARDTHAT WELL-KNOWN FLYING
```

```
14510 DATACAKE TIN THE USS SPLIT INFINITIVE
.  UNFORTUNATELY THERE  DOESN'T
14520 DATASEEM TO BE ANYONE AROUND SO     I
F YOU GET SCARED YOU'LL HAVE
14530 DATANO-ONE TO 'KLING ON' TO!!!
14600 DATAYOU ARE IN A SMALL CAVERN.  THE W
ALLS   AND CEILING ARE COVERED WITH
14610 DATACOBWEBS ANDTHE ONLY LIGHT COMES F
ROM TWO BLAZING   TORCHES.
14620 DATAIN THE DUST ON THE FLOOR ARE   SE
VERAL SETS OF STRANGE TRACKS
14630 DATASUCH AS  MIGHT BE MADE BY A GIANT
 INSECT.
14700 DATA"YOU HAVE ENTERED A LIBRARY FULL
OF WELL-MADE LEATHER FURNITURE,"
14710 DATA"ROWS AND ROWS OFBOOKS, AND A DEA
D BODY LEFT OVER FROM A MURDER"
14720 DATAADVENTURE.  A PICTURE ON THE WEST
WALL HAS BEEN PULLED BACK
14730 DATATO REVEAL A   SMALL SAFE.  THE SA
FE IS STILL CLOSED.
14800 DATAYOU ARE IN A FOREST THAT DOESN'T
SEEM TOHAVE AN END.  ITS
14810 DATAQUITE A NICE FOREST - AS FORESTS
GO - BUT THAT'S THE BEST THAT
14820 DATATHAT COULD BE SAID FOR IT.  STILL
 THE PATH MUST LEAD SOMEWHERE
14830 DATABECAUSE THERE'S A BROKEN SIGN TO
THE EAST THAT SAYS 'TO GO'
14900 DATAYOU SEEM TO BE ON A DESERT ISLAND
 -     THERE'S NOTHING BUT SAND SEA
14910 DATAAND A FEW  PALM TREES AS FAR AS T
HE EYE CAN SEE.   THERE DOESN'T
14920 DATASEEM TO BE MUCH GOING ON  HERE (T
HOUGH YOU COULD
14930 DATATRY DIGGING - IF YOU HAVE A SPADE
).
15000 DATAWELCOME TO GOBBETANIA - THE LAND
THAT   TIME FORGOT.
15010 DATAYOU ARE GREETED BY A SMALL FIGURE
 WITH A LARGE SWORD - IT'S
15020 DATABULBOUSFAGEND  (WHO CAN REINCARNA
TE AT WILL!).
15030 DATABUT WHY IS HE WAVING THAT SWORD A
T YOU?  I THINK HE WANTS TO FIGHT!!
```

```
151ØØ DATAAS YOU ENTER A DAMP AND GLOOMY CR
YPT A  BAT
1511Ø DATAFLIES PAST YOU INTO THE SHADOWS.
 INTHE FAR CORNER THE LID OF A
1512Ø DATACOFFIN OPENS - A SMILING GENTLEMA
N WITH LONG TEETH  AND AN EVEN
1513Ø DATALONGER BLACK CLOAK STEPS OUTAND M
OVES SILENTLY TOWARDS YOU.
152ØØ DATAAT THE TOP OF THE STAIRS YOU FIND
 YOUR- SELF IN
1521Ø DATATHE COCKPIT OF A JUMBO JET.  ONEL
OOK AT THE CONTROL PANEL TELLS YOU
1522Ø DATATHATTHE PLANE IS NEARLY OUT OF FU
EL  -  AND GOING INTO A STEEP DIVE.
1523Ø DATAI'D SAY YOU'D  STEPPED INTO THE W
RONG ADVENTURE!!!
153ØØ DATAO.K. - HERE WE GO THEN. YOU STEP
THROUGHTHE DOOR INTO THE ROOM
1531Ø DATADEDICATED TO A   NEW GAME CALLED
'APOCALYPSE YESTERDAY'.  UNFORTUNATELY
1532Ø DATASOMETHING HAS GONE WRONG - THE RO
OM IS A NUCLEAR WASTELAND - AND THERE
1533Ø DATAIS NO WAY OUT!!!
15999 REM ********  INTRODUCTION
16ØØØ PRINT"[CLR][DOWN * 3][RVS][CTRL 6] (1
1 SPACES) THE CASE OF THE (22 SPACES) ";
1601Ø PRINT"[CTRL 6][RVS]  MISSING ADVENTUR
E (13 SPACES) "
1602Ø PRINT"[CTRL 2][DOWN]  IT WAS A DARK A
ND GLOOMY DAY AND THE  RAIN WAS ";
16025 PRINT"COMING DOWN IN ";
1603Ø PRINT"SHEETS - COTTON SHEETS, SATIN S
HEETS - I SHOULD HAVE    STAYED IN BED!"
1604Ø PRINT"[DOWN]  I'D SPENT SEVERAL DAYS
TRYING TO TRACKDOWN EDDY THE KWILL,  THE";
1605Ø PRINT" ULTIMATE IN   UNDESIRABLE CHAR
ACTERS.  I'D ENDED UP ATTHE WRONG END ";
1606Ø PRINT"OF A BLIND ALLEY.":PRINT"[DOWN]
SUDDENLY A DOOR AT THE END OF THE ALLEY ";
1607Ø PRINT"OPENED.  A HAND BECKONED TO ME,
 TO BE   HONEST I WASN'T SURE WHAT TO DO."
1608Ø GOSUB13ØØØ
1609Ø PRINT"[DOWN]  I THINK I'LL LEAVE THE
CHOICE TO YOU. PRESS D AND I GO THROUGH ";
161ØØ PRINT"THE DOOR.":PRINT"PRESS ANY OTHE
R KEY AND I QUIT THE CASE.":INPUTZ$
```

```
16110 IFZ$<>"D"THENPRINT"[CLR][DOWN * 3]O.K
. - I QUIT.":END
16120 PRINT"[CLR][DOWN * 3]GOOD CHOICE.  NO
W I'M YOU AND YOU'RE ME.";
16125 PRINT"FROM NOW ON ITS YOUR CASE."
16130 PRINT"GOOD LUCK - YOU'RE GOING TO NEE
D IT!!":RETURN

READY.
```

Index

adventure, location of, 51–70
adventure groups, 189
  networks, 191
arrays,
    Program 5.8, 94
    Program 5.9, 97
  check routines, 96
  dimensions, 71
    Fig. 5.1, 72
    Fig. 5.2, 72
  zero elements, 73
    Program 5.1, 74

booby-traps, 103
    Program 5.10, 103
brainstorming, 13

cast list, 20
characters,
  computer-set, 38
    Program 3.4, 39
  fixed, 21
    Program 3.1, 27
  fixed with options, 29
    Program 3.2, 30
  hero/heroine, 23
  progressive, 21
  ratings,
    health, 83
    height and weight, 85
    intelligence, 84
    luck, 85
    skill, 84
    strength, 74
    wealth, 85
  second level, 25
  status display, 43
    Program 3.5, 44
  super-heroes, 25
  third level, 26
  user-controlled, 33
    Program 3.3, 34
combat, 47
    Program 3.6, 48
commands,
  documenting, 159
  drop,
    Program 5.4, 82
  get,
    Program 5.2, 76
    Program 5.3, 79
  inventory, 75
  parsing, 5
    Program 8.1, 147
  parsing, compound, 5
    Program 8.2, 153
  save game, 98
comprehensibility, 7

delimiters, 5
documentation,
  block diagrams, 162
    Fig. 9.1, 163
  commands, 159
  flow charts, 164
    Fig. 9.2, 164
  program layout, 169
  variables, 166

education, 192

graphics, 172–187
  sprite grid,
    Fig. 10.1, 180
  sprite sketch,
    Fig. 10.2, 181
  sprites, 174
    Program 10.1, 176
  use of, 172
  VIC map, 174

ideas list, 13
interest, 8
interior decor, 71–105
internal consistency, 8, 99

language, 145–160, 194
  Inglish, 5
  Interlogic, 6
line header block, 116
    Fig. 6.1, 116

map-making, 57
maps,
  boxes and lines, 58
    Fig. 4.2, 59
  colour coding, 66
  linked octagons, 64
    Fig. 4.5, 66
    Fig. 4.6, 67
  linked squares, 59
    Fig. 4.3, 60
    Fig. 4.4, 62
  linked squares, calculated moves, 60
    106
    Program 4.1, 61
    Program 4.2, 62
movement codes, 106–124
    Program 6.1, 108
    Program 6.2, 112

PEEKs and POKEs, 115
    Fig. 6.2, 117
    Fig. 6.3, 118
    Fig. 6.4, 119
plot, 7, 10–19
'pointers', 115
problem setting, 8, 103
program organisation, 161–171
programming, 166
  GOTO/GOSUB, 167
  loops, 168
  REM, 167
  speed tips, 171

RAM space, 53, 193
  chart, 56
room contents, 86, 101
  random items, 87
    Program 5.5, 87
    Program 5.6, 90
    Program 5.7, 92
room descriptions, 6

sounds, 188–198
  SID, 181
  SID, map, 183
    Program 184, 185, 186
storyboard, 15
  examples, 16, 17, 18
storyline, 12

text packing, 125–144
  cassettes, 128, 193
  decode,
    Program 7.4, 141
    Program 7.5, 142
  disk storage, 127, 193
  encode,
    Program 7.1, 130
    Program 7.3, 134
  string analysis,
    Program 7.2, 132

variable list table (VLT), 121

zero page, 115

CREATE YOUR OWN ADVENTURE GAMES!

Are you an experienced programmer or new to computing? Do you want to write games for profit or just to bamboozle your friends? If you answered 'YES' to any of these questions and you are interested in adventure games then you will enjoy this book. Your Commodore 64 with its large memory and stunning graphics, is an ideal computer for adventure games.

In this book you will find every aspect of adventure writing – from stories and character creation to the more technical programming tricks found in top commercial games – described in detail. The only boundaries in the world of the computer adventure game are those set by your own inventiveness, imagination and skill. Numerous examples and programs which can be incorporated into your own adventure games are included to help you push those boundaries to the limit.

The Author
A. J. Bradbury, who also writes under the name John Noad, is a computer studies teacher and a keen adventurer. His articles have appeared in *Windfall* and *Personal Computer News*.

More books on the Commodore 64 from Granada

COMMODORE 64 COMPUTING
Ian Sinclair
0 246 12030 4

THE COMMODORE 64 GAMES BOOK
Owen Bishop
0 246 12258 7

SOFTWARE 64
Practical Programs for the Commodore 64
Owen Bishop
0 246 12266 8

INTRODUCING COMMODORE 64 MACHINE CODE
Ian Sinclair
0 246 12338 9

COMMODORE 64 GRAPHICS AND SOUND
Steve Money
0 246 12342 7

Front cover illustration by Angus McKie

GRANADA PUBLISHING
Printed in Great Britain 0 246 12412 1

IN CANADA
$14.95
COLLIN S.



This was brought to you

from the archives of

http://retro-commodore.eu