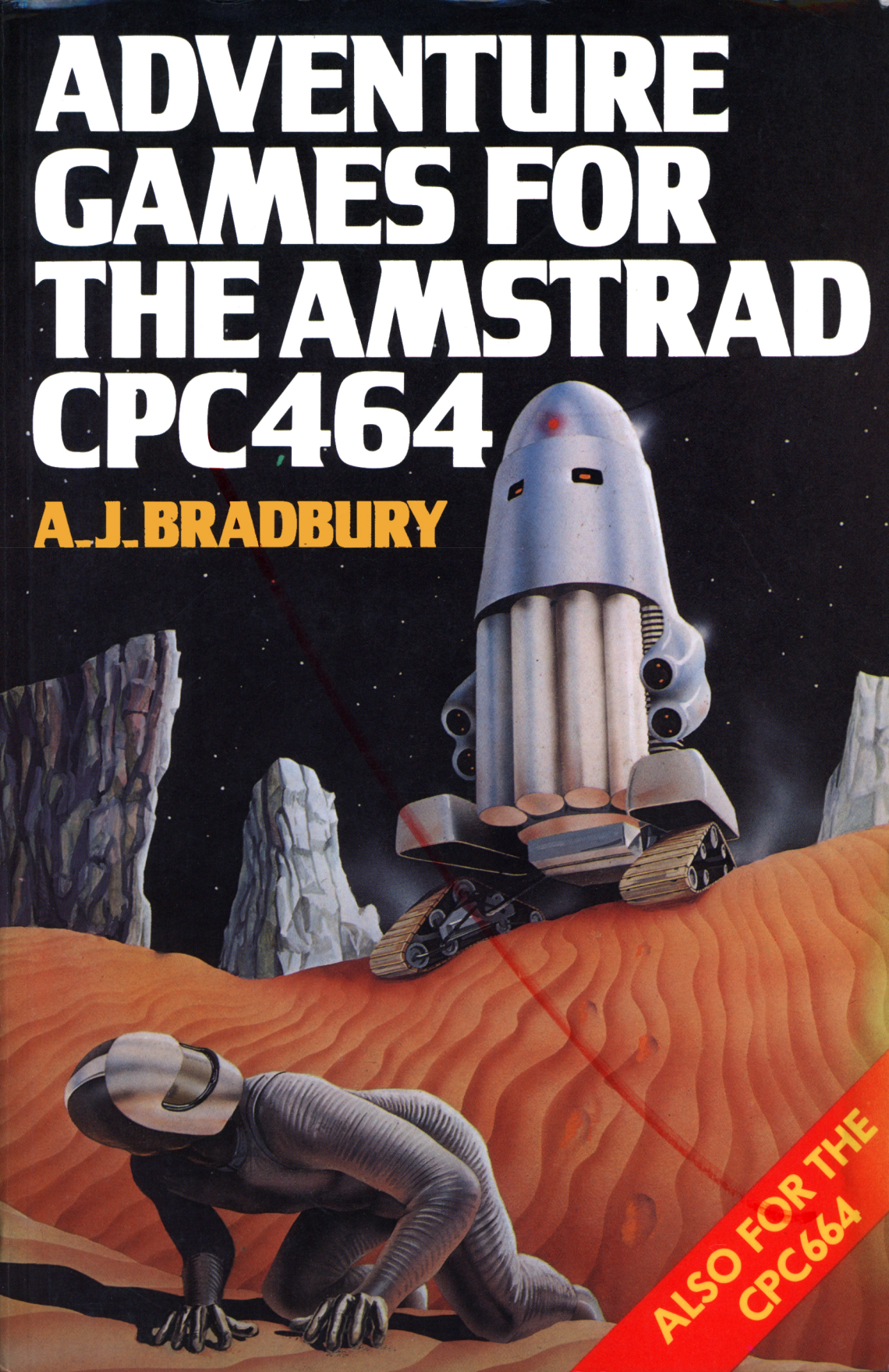
ADVENTURE GAMES FOR THE AMSTRAD CPC464

A. J. BRADBURY



ALSO FOR THE CPC664

Adventure Games for the Amstrad CPC464

Other books for Amstrad users

Amstrad Computing
Ian Sinclair
0 00 383120 5

Introducing Amstrad CPC464 Machine Code
Ian Sinclair
0 00 383079 9

Filing Systems and Databases for the Amstrad CPC464
A.P. Stephenson and D.J. Stephenson
0 00 383102 7

Practical Programs for the Amstrad CPC464
Audrey Bishop and Owen Bishop
0 00 383082 9

Sensational Games for the Amstrad CPC464 and CPC664
Jim Gregory
0 00 383121 3

40 Educational Games for the Amstrad CPC464
Vince Apps
0 00 383119 1

The Amstrad CPC464 Disc System
Ian Sinclair
0 00 383177 9

Adventure Games for the Amstrad CPC464

A.J. Bradbury

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985
Reprinted 1985

Distributed in the United States of America
by Sheridan House, Inc.

British Library Cataloguing in Publication Data
Bradbury, A. J.
Adventure games for the Amstrad CPC464.
1. Computer games 2. Amstrad computer——
Programming
I. Title
794.8'028'5404 GV1469.2

ISBN 0-00-383078-0

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

Contents

Important Note to the Reader

All of the programs in this book were tested on an Amstrad CPC464 before being LISTED for publication. If you are unable to get any of the programs to work on your own machine please check that you have copied the original listing *accurately*.

The form of BASIC used for the Amstrad - LOCOMOTIVE BASIC - though slightly slower than BBC BASIC, is arguably superior to any other form of micro-BASIC. Indeed, it offers several advanced features found on no other micro (at the time of writing).

However, the very fact that LOCOMOTIVE BASIC is so advanced may offer a few problems to newcomers to computing. For this reason this book includes a chapter (Chapter 3) which deals specifically with the range of instructions you are most likely to need when writing an adventure. If you are not already familiar with LOCOMOTIVE BASIC, or if there is anything in the programs that is not entirely clear, read Chapter 3 carefully.

All programs have been written using Mode 1 (i.e. with the 40 column screen display).

Owners of the Amstrad CPC664 may note that all the programs in this book are written in Version 1.0 of LOCOMOTIVE BASIC (for the CPC464) and run equally well on either machine.

Chapter One
Once Upon a Time...

READ MESSAGE

... Geresfae, last of the Rangers of Adarorn, paused beside one of the rocks dotted around the desert landscape and read again Nmosoc's last, cryptic message: 'Nearer the 5th moon than the 3rd – if the Zeebnoi are passed then the Arc will be yours'...

LOOK

... The Ranger looked up to the five moons stretching across the deep-hued sky. The 3rd moon had to be the one in the middle no matter how you looked at it. But which was the 5th moon – should he go North or South?...

GO NORTH

... With nothing to guide him Geresfae turned northwards across the rippling sands. He wondered, vaguely, what Zeebnoi was, or were. Things or one thing, beings or one being? His thoughts were suddenly interrupted by a dull, rumbling sound...

DOWN, LISTEN

... Puzzled, Geresfae knelt down to put his ear to the sand – then realised that he had his helmet on. Still the rumbling seemed much louder now, and closer!...

LOOK SOUTH

... Without rising Geresfae turned, and saw something coming over the dunes towards him, moving smoothly but noisily on caterpillar tracks. Geresfae started up. Was this a Zeebnoi? And if it was what should he do – stand or run?...

Could you write adventures like this, full of thrills, danger and high adventure? If you've never tried before you may not think you can. But

read on – it really isn't 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, in its own area, unique.

O.K. – that's the sort of claim you'd expect an author to make about his book. But this time it really is 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 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 (who dwelt in the misty halls of the palace 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 player's 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 already 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 had concentrated on producing home micro versions and variations of

arcade machine originals. As a result the adventure market has remained almost 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 been copied in a number 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, 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 empty promise 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 its 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 and 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 affecting 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 inconsistent, set of commands.

In Chapter 9 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 by many 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 six 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 8 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 these 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 complicated 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 £10–£20. 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 toward 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 one 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 5 you'll find descriptions of the various ways of mapping out an adventure with their advantages and disadvantages. In Chapter 7 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 to be borne in mind 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 I am using the word *plot* 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 the 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 the 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 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 books? 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 that 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 that you already know about, one that interests you (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 in 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 it's 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) – and 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 note down all the characters, locations and objects you would need to include in your game if you used every single 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 other 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 about how many changes you make! It can be tempting, when a really good idea comes along, to rewrite your previous storyline so as 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 is, therefore, 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 moves 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 11.)

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. Also 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 he can find it.

Version 3: Screen 24. Laboratory.
With a quiet hiss the steel door slides silently 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

The Computer at Your Command

In the next chapter we will begin to examine some of the pieces, or *modules*, that go together to make an adventure game. In other words, we will actually begin to write a program. But coding an adventure game isn't the easiest thing in the world to do – unless we come to the job fully prepared. In this chapter, therefore, I shall quickly go over some of the most useful commands that can be found in the Amstrad's extremely powerful version of the BASIC computer language. If you are *sure* that you already understand LOCO' BASIC then by all means go straight on to Chapter 4, but if you have *any* doubts I strongly suggest that you go through this chapter, trying out the various short routines, before continuing with the rest of the book.

In order of appearance ...

The commands I have chosen to study here are, of course, all described in the Amstrad User Guide, an extremely well-prepared manual. If there is anything that is not clear to you as you read this chapter I recommend that you look up the command in question in the User Guide, Chapter 8.

The instructions that I want to deal with are given below in alphabetical order.

AND The main purpose of the AND command in BASIC is to control the program by ensuring that certain variables are *all* in the correct state. That is to say, operation C is controlled by the condition of variables A and B. For example:

```
100 IF A < 10 AND B <> 12 THEN GOSUB
    2000
```

In this example the program will *not* go to line 2000 if (a) A is equal to or greater than 10, (b) A is less than 10 but B *is* equal to 12, or (c) if A is equal to or greater than 10 *and* B = 12. In other words, both conditions (or all three or four conditions – you can use more than one AND in a line) *must* be 'true' before the command which follows the AND can be executed.

The AND instruction can be extremely useful in an adventure game for making sure that, for example, the player has not taken a short cut or that he has not put down objects that you want him to carry with him. You might decide that the player needs a sword *and* a shield before he can fight a certain monster, so you check the 'object array' (see Chapter 6) to see that he has *both* objects when he meets the monster. If he hasn't, the monster automatically wins the fight!

```
100 DIM OB$(10,2)
110 OB$(5,1) = "-1": OB$(7,1) = "-1"
120 IF  OB$(5,1) = "-1" AND OB$(7,1) = "-1
    THEN 200 ELSE 300
200 PRINT  "YOU HAVE KILLED THE DRAGON!!!"
    END
300 PRINT "YOU ARE NOT PROPERLY ARMED - THI
    DRAGON HAS KILLED YOU!!"
```

Try changing the values inside the inverted commas in line 110 and see what happens.

CHAIN/CHAIN MERGE The CHAIN command is most often used simply to LOAD a program, but it can also be used to LOAD another program over the top of the program already in memory. This is particularly useful if you want to make a large adventure out of several smaller games. Enter the first program below and SAVE it as PROG2. Then enter the second program and RUN it:

```
10 REM  -  To be SAVEd for demonstration
   CHAIN
20 PRINT "Hello - I'm PROG2"
30 END
```

When you've saved this program don't forget to *rewind* the tape before you RUN the second program.

```
10 REM - Enter me then RUN me
20 PRINT "Hello - I'm PROG1"
30 CHAIN "PROG2"
40 PRINT "Hello - PROG1 is back again!"
```

Note: Although I've included a line 40 in PROG1 this is only to demonstrate the fact that when you CHAIN one program from another the first program is wiped out by the second. To see how the CHAIN MERGE command varies from a plain CHAIN instruction can be demonstrated by re-using the two programs above with a single amendment to line 30 of the PROG1 program. That line should now read: 30 CHAIN MERGE "PROG2".

You will notice, by the way, that CHAIN and CHAIN MERGE are the equivalent of the RUN command in their effect (i.e. the nominated program is both LOADed and RUN without further user action being required).

CHR$ This instruction has two main uses – to allow use of control characters (see section 9.1 of the User Manual), and to allow use of characters 123 to 255 of the Amstrad character set (see Appendix 3 of the User Manual), including any user-defined characters.

CLS This instruction is mainly used when you want to present one or more full screen displays without having any text left over from previous displays. We will see this command in action several times in the programs in the next chapter. CLS can also be used to alter the PAPER colour in just one operation. This can be clearly seen in Program 11.1.

DATA A large part of any adventure game is taken up by shuffling items around in arrays (see Chapter 6). In order to make it easy to find, and easy to use, we might store such information at the *end* of the program in DATA statements. This also allows the program to RUN faster than if our DATA were mixed up with the rest of the program. The DATA statement is *always* used together with the READ command, and may also be affected by the RESTORE command:

```
10 READ A$,B$,C$,D$
20 PRINT A$
30 PRINT B$
40 PRINT C$
50 RESTORE 210
60 READ E$
```

```
70 PRINT A$;" ";B$;" ";C$;" ";E$: END
200 DATA HERE,ARE,FOUR
210 DATA WORDS
```

You can see here how the RESTORE 210 command allows us to use WORDS twice, even though it only appears *once* in the DATA. Compared to the more usual RESTORE command which RESTOREs *all* DATA, the ability to RESTORE particular pieces of DATA is especially useful as it means that we can cut down the total number of DATA statements and RESTORE just those items which we need to use more than once.

Note: If, at any time, you should happen to mis-write a line of DATA so that, for example, 'string' data is READ to a 'numerical' variable then the error message will not be TYPE MISMATCH, as the User Manual states. Instead you will get SYNTAX ERROR and the erroneous DATA statement will be presented for editing. So if the computer does throw up a DATA line for editing and you can't see anything wrong, do check the accompanying READ instructions before tearing your hair out.

DIM Some computers allow you to use an array of less than 11 elements without DIMming it first (that is, without telling the computer that you are going to use an array). This is true of the Amstrad. But arrays should be DIMmed before they are used anyway, because using arrays carelessly can waste a lot of space and create some very unpleasant problems which may be hard to spot before they actually ruin your program. We will be dealing with arrays in detail in Chapter 6.

ELSE This statement may not look very exciting but it is one of the items which make LOCO' BASIC so advanced compared with most other versions of BASIC. The main value of ELSE lies in the fact that it allows you to combine two instructions on a single line (see the section on AND, above). Thus we can write:

```
10 IF B = 10 THEN 100 ELSE 200
```

where in most other forms of BASIC we would have to write:

```
10 IF B = 10 THEN 100
20 GOTO 200
```

A major feature of adventure games is that the player, and the program, have to make many, many choices about what is to happen

next. You will soon discover that the ELSE statement is very useful indeed!

FOR As I have remarked elsewhere in this book, it can often seem that an adventure game program is nothing more than a gigantic series of FOR ... NEXT loops. This isn't really true, of course, though loops do play an important part in adventure programs. The function of a loop – either FOR ... NEXT or WHILE ... WEND is to control how often something occurs. A good example of this can be found in the 'combination lock' routine at the end of Chapter 6 where the player is allowed only 10 attempts at opening the safe. The FOR ... NEXT loop runs like this:

```
100 SAFE%=RND*100
110 FOR X=1 TO 10
120 INPUT A
130 IF A=SAFE% THEN X=10:NEXT:GOTO 300
140 NEXT (OR NEXT X)
150 PRINT"Hard Luck - the safe stays
    locked!":END
300 PRINT"Congratulations - you've
    unlocked the safe!!!":END
```

Now, if you have already entered and RUN this little program, you will know what I know, what I found out after one rather confusing late-night session – this program doesn't work!

So why present a program that doesn't fulfil it's purpose? The answer is that the program I'm about to describe is almost unique in microcomputing. You see, the Amstrad will actually search ahead during a FOR ... NEXT loop to find out where the loop ends. Just how it does this is a bit of a mystery – since it seems to simply reckon up the location of the nearest NEXT command, without taking note of the context within which it occurs. This means that the two NEXT commands in the program above look to the Amstrad as though they are both free standing NEXT commands applying to the same FOR (which would, if true, be illegal).

In short, then, LOCO' BASIC does not allow us to 'close' a loop when we drop out before completion. In order to make this program work we would need to amend line 130 to read

130 IF A = SAFE% THEN FLAG = 1

or

```
130 IF A = SAFE% THEN 300
```

In the first case the loop would have to be completed before anything else could happen, and an additional line must then be added which will read the value of the 'flag' (see section on the IF command):

```
145 IF FLAG = 1 THEN 300
```

The second alternative allows us to drop out of the loop as soon as the input (A) equals the required value (SAFE), if it ever does, but the loop is left open – a less than wholly satisfactory situation. For all practical purposes, then, I would recommend that the FOR ... NEXT loop be avoided wherever you may wish to drop out of the loop before completion. If you turn to the program I've already mentioned (at the end of Chapter 6) you will see that I have achieved exactly the same effect by replacing FOR ... NEXT with the WHILE ... WEND form of loop.

Program execution controllers

One of the things that makes a computer program particularly hard to read is the fact that programs often do not operate, or 'execute', in a straightforward line-by-line manner. The lines of a program may be *numbered* 10, 20, 30, 40, and so on, but that doesn't necessarily mean that the computer will execute line 10, then line 20, then line 30, etc. The order in which the lines are actually executed will depend upon the contents of each line.

If we only wanted to perform one or two fairly simple operations we would almost certainly be able to achieve this aim with a program in which each line was executed just once and in numerical order. Few programs, however, are that simple so we need some other means of moving from one place to another within a program without executing the lines in between as we do so. In practice, LOCO' BASIC offers two different means of carrying out such an operation: GOTO and GOSUB.

GOTO When we want to make a jump in the course of a program the first thing that has to be decided is whether it is to be a 'one way journey' or a 'round trip', so to speak. Take the following example:

```
500 PRINT "YOU ARE HOLDING: "
510 GOTO 900
520 PRINT K$
530 END
```

The one thing we can be sure of here is that unless another instruction tells the computer to go back to line 520 or line 530 these lines will never be used. Why not? Because the GOTO instruction is strictly 'one way'. When the computer performs a GOTO jump it does nothing about saving the address that it was sent *from* (see Chapter 10), and therefore it cannot 'return' unless it is specifically sent back to execute those other two lines.

Generally speaking, a GOTO instruction need only be used when you want to execute a section of your program which has an unpredictable outcome or, in the case of an adventure program, when you 'want' to make the program as hard to read as you can.

GOSUB The most direct alternative to the GOTO statement – shared by most other varieties of BASIC – is the GOSUB command. To repeat the programming example above:

```
500 PRINT "YOU ARE HOLDING: "
510 GOSUB 900
520 PRINT K$
530 END
```

In this format we can safely say that lines 520 and 530 *will* be executed unless we deliberately prevent it. Thus the correct 'flow' of the program will be 500 – 510 – 900 – 520 – 530. If line 510 had been a multiple statement line:

```
510 GOSUB  900:  PRINT "A TOTAL WEIGHT OF
    ;IN;" LBS."
520 PRINT K$
530 END
```

the program would return to execute the PRINT statement in line 510 and then move on to lines 520 and 530.

In most versions of BASIC the GOSUB command is used to

execute *any* subroutine where it is known that the subroutine will execute successfully. To show what I mean let's take a concrete example such as the 'command parsing' routine (described in detail in Chapter 9).

The first thing that a command parsing routine needs to do is ensure that each command makes sense, and if it does the program can move on to check whether a particular command can be executed. DROP THE SWORD would make no sense at all to the computer if it were entered as DRIP THE SWORD, and even if the command is typed in correctly it cannot be executed unless the player actually *has* a sword which he can drop.

So let's suppose that the player has entered GO NORTH. Once the program has received this command its first two operations – checking for a valid command and checking whether the command can be executed – are both capable of producing completely opposite results, success or failure. This means that we must use the GOTO instruction both times since we cannot know in advance whether the program will be able to carry out the command or whether it will have to present an error message and go back for a fresh command.

If both operations are completed successfully the next move is to relocate the player according to his chosen move. This time we *know* that the operation is bound to be successful, even though the exact result will depend on where the player was before he moved, so in this case we can use the GOSUB command.

IF We have already seen the IF instruction at work in several previous examples (see AND, ELSE, etc.) but it is worth mentioning it again in connection with what are known as 'flags'.

Flags are nothing whatever to do with computerised parades – instead, they are variables whose sole purpose is to show whether a certain condition exists or not. For example, they may either be set to signal an 'ON/OFF' situation – the equivalent of YES/NO – using 1 for YES and 0 for NO, or they may be used as accumulators to show whether a given operation (or set of operations) has been carried out the required number of times. A typical example of an ON/OFF variable can be seen in the way that we use part of the Object array described in Chapter 6.

Without going into too much detail here I will just say that a careful check *must* be kept on the location and status of all the objects used in an adventure game. So one of the variables attached to each object is used as a 'location marker' and can have one of three values: −1 (minus 1), which means that the player is carrying that object; 0 (zero), which

means that the object is no longer available for some reason; or any positive value (1 upwards) which is taken to be its present location – that is to say, it *is* available but the player isn't carrying it at the moment and therefore cannot use it.

This may not look like a YES/NO situation – until we put it in a practical setting. (*Note:* In the following examples CH is the object's position in a list of objects, OB$(CH,0) will be the name of the object, OB$(CH,1) will be the 'flag' value for that object, and N$ is the object name entered by the player as part of a command.)

(a) If we want to pick an object up we need to know whether we already have it – in other words does the flag equal −1? If it does, then obviously we can't GET that object again:

```
460 IF  OB$(CH,1)  =  "-1"  THEN PRINT "YOU
    ALREADY HAVE THE ";N$;"!"
```

(b) If the flag isn't set to −1 then maybe the object is no longer available (it might be something that can only be used once):

```
470 IF  OB$(CH,1) = "0" THEN PRINT "SORRY -
    THE ";N$;" ISN'T AVAILABLE."
```

(c) If we're still in business after the last two checks – both simple YES/NO choices, you notice – then all we need to know is whether the required object is in the same room as the player (i.e. the flag must have the same value as PL – Player's Location):

```
480 IF  VAL  (OB$(CH,1))  <>  PL THEN PRINT
    "SORRY  =  THE  ";N$;"  ISN'T  HERE."
```

The <> symbol means 'not equal to', of course, and though we're doing a 'negative comparison' here we are still looking for a simple YES/NO answer.

To summarise this section, the IF statement (usually in the form IF ... THEN) has a variety of uses in any program. On top of its more common functions, it comes in extremely useful in adventure games as a quick and easy means of checking the current status of any situation. Not surprisingly, then, we will be meeting the IF statement many, many times in the program modules throughout this book.

INKEY (value) The purpose of the INKEY instruction is to test for a specific key press. This involves using the set of key values shown on the keyboard illustration on page 16 of Appendix III of the User Manual.

```
10 IF INKEY(69)=32 THEN 20 ELSE 10
```

This example would halt program execution until both the <SHIFT> key and 'A' are pressed (i.e. to give capital A).

Note: The values used to define keys/characters used in an INKEY() instruction are not the ASCII values (i.e. those shown in Appendix III of the User Manual). The INKEY instruction may *not* be used to collect just any key press as a statement like X=INKEY is illegal.

**INKEY$** Where INKEY collects a numerical value for a key press, the INKEY$ statement will collect a single character which it will hold in string form:

```
10 X$ = INKEY$
```

For those readers who are already familiar with another form of BASIC, I should point out that the INKEY$ command is the *only* substitute for both GET and GET$. When used to collect a numerical value, therefore, the resulting key press must be converted to a number using the VAL() command.

INPUT/LINE INPUT The last of the statements used to collect information from the keyboard is INPUT. This is the *only* way of collecting more than one character in a single operation – though this is not always an advantage.

(a) Firstly, the INPUT statement is the only one in the group where the end of the input must be indicated (by the user) by pressing the <RETURN> key. So if you're going to use INPUT, be sure to indicate to the player that he *is* expected to press the <ENTER> key when his entry is complete.

(b) Secondly, an INPUT item may contain any characters at all (so long as you are INPUTting to a *string* variable – e.g. X$) *except* a comma. If a comma is included then, unless the input starts and ends with inverted commas, everything from the comma onwards will be ignored.

(c) Where INPUT is coupled with a numerical variable (e.g. X or X%) then *only* numerical input will be accepted:

```
10 INPUT X
```

If you entered 123 in response to this line then X would equal 123. If you entered 123ABC X would still be set to 123 and everything from A onwards would be ignored. If you just entered ABC then X would equal 0 as none of the input would be valid as a number.

One way of beating the 'comma rule' is to use INKEY$ and *string concatenation* (making one big string out of lots of little strings):

```
110 T=0
120 WHILE A$<>CHR$(13) AND T<200
130 A$=INKEY$: IF A$="" THEN 130
140 B$=B$+A$: T=T+1
150 WEND
160 IF A$=CHR$(13) THEN B$=LEFT$(B$,
    LEN(B$)-1)
```

The maximum value for T in this routine will be 255 as the program will automatically 'crash' if you try to build a string of 256 characters or more. CHR$(13) is the equivalent of the <ENTER> key. Since this method involves adding the <ENTER> key press on to the concatenated string, line 160 is designed to take it out again if it is the last character received.

A far easier method is provided only in LOCO' BASIC (though it is included in MICROSOFT BASIC on which most forms of micro-BASIC are based). By using the LINE INPUT command, instead of the plain INPUT the Amstrad will accept everything that you enter, including commas (a normal INPUT command will ignore everything after and including a comma).

By the way, the INPUT statement is the only 'collection' statement which can include a printed display:

```
230 INPUT "PLEASE ENTER A NAME FOR YOUR
    CHARACTER AND PRESS <RETURN>: ";NA$
```

Also note that the semicolon between the message to be displayed and the variable name does serve a purpose. In a simple statement like INPUT NA$ the computer automatically displays a question mark

where the input will commence. If text is included in the INPUT statement then the question mark will only be displayed if the semicolon is included. So if you don't want a question mark use a comma instead.

INSTR From the adventure programmer's point of view, this instruction is probably the most useful command to be found in Amstrad BASIC. In its longer form – INSTR (long string,short string,number) – it allows you to search for any string within a longer string, starting at any appropriate position.

Its most practical purpose, from our point of view, simply involves checking whether a shorter string occurs *anywhere* within a longer string in order to check for valid command input. I will be dealing with this at some length in Chapter 9 so I'll just give a very brief example here from what I call the '2 Letter Command Parser'. In this routine all valid verbs and nouns are coded using one letter for each. To find out whether the first letter in the command (i.e. the verb) is valid, all we need is:

```
80 VE$="ABCDEFGHIJ"
90 V$=INKEY$:IF V$="" THEN 90
100 VP=INSTR(VE$,V$)
```

To perform this same search in standard BASIC takes four or five lines including a FOR ... NEXT loop. All we need to do here is to find out if VP is greater than 0 and if it is we have a valid verb and can go on to collect a noun code. If VP equals 0, then the verb code wasn't valid and we send the program back for fresh input. This method is not only quicker (no loop) and easier, but also less costly in terms of programming space.

INT/CINT Put very simply the INT (or INTeger) function is used to round numbers off. It always rounds *down*, however, never up, so INT(3.0009) and INT(3.9999) will both give the value 3. In most other versions of BASIC the INT function is frequently used together with the RND (RaNDom) function to give a random whole number. However, as you've probably already noticed, LOCO' BASIC offers several variations on this theme. Where INT will always round a number down, FIX will round a number towards 0 (i.e. negative numbers will actually be rounded up), and CINT will round a number off to the *nearest* whole number. You may also assign a number to an 'integer variable' such as B%. Normally this will have the same effect as using the INT command, though when combined

with the RND command (which I'll deal with later on) you may get a rather different result.

**LEFT$** Like MID$ and RIGHT$ (see below), the LEFT$ function is made partially redundant by the presence of the INSTR function. Nevertheless, it is perhaps the most useful of the three $ handlers as it allows us to shorten input commands for use in comparisons. Most verbs and nouns can be distinguished by just the first two or three letters – DROP becomes DRO, GET is GET, TAKE becomes TAK, KILL becomes KIL, and so on:

```
80 VE$ = "DROGETTAKKIL....."
90 INPUT "WHAT NOW? "CO$
100 V$ = LEFT$(CO$,3)
110 VP = INSTR(VE$,V$)
```

Note: The LEFT$ command always collects a given number of characters from a string starting with the *first* letter on the left. If you want to take the letters from anywhere else in the string you must use MID$ or RIGHT$.

LEN This is probably one of the least-used functions in adventure programming but it, too, has a purpose in more complicated command parsing routines. Since this will become clear in Chapter 9, I shall only mention here that LEN(X$) may be used to find the number of characters – *including* blank spaces – in any string be it text or a string variable:

```
10 X$ = "HELLO THERE"
20 A = LEN("HELLO THERE")
30 B = LEN(X$)
40 PRINT A,B
50 PRINT LEN("HELLO THERE"),LEN(X$)
```

This brief routine will give the value 11 four times over.

**MID$** The MID$ function, like LEFT$, is used to isolate part of a longer string. As in other forms of BASIC the Amstrad MID$

function is usually given *three* 'parameters' to work with – the text or string variable to be examined, the position of the first letter (numbering from the left) to be dealt with, and the number of other letters (working to the right) to be handled. Where the Amstrad MID$ function varies from many other versions of the command is at the point where only two parameters – the text/string and one number – are given. In other forms of BASIC the lone number is often taken to represent the number of letters to be handled, so that MID$ becomes the equivalent of LEFT$. In LOCO' BASIC, however, a single number parameter is taken as the number of the first letter to be handled, so the result will be a printout or a string variable with all the letters in the larger string from the numbered character up to the right-hand end of the string.

```
10 A$ = "ADVENTURE"
20 PRINT MID$(A$,3)
30 PRINT MID$(A$,3,4)
40 PRINT MID$(A$,5,1)
```

NEXT See FOR above.

NOT The basic purpose of this operator is to *reverse* the result or meaning of any statement or calculation in terms of TRUE and FALSE. On the Amstrad, any zero result is read as 'false', and any other result (positive or negative) is taken as 'true'. Likewise, you can force a 0 or −1 (the value of TRUE) by testing a situation in the following manner:

```
10 X = 5
20 PRINT X = 6: REM  = FALSE/0
30 PRINT X = 5: REM  = TRUE/-1
40 PRINT NOT X = 6: REM  = TRUE
50 PRINT NOT X = 5: REM  = FALSE
```

Run this short routine to see exactly how the TRUE/FALSE conditions and the NOT operator are handled by the Amstrad.

ON See GOTO/GOSUB above.

OR You will remember that we used the AND operator to test for

two (or more) conditions which agreed with our expectations. For an AND statement to succeed each and every substatement must succeed. When the substatements are linked by the OR operator, however, if *any* substatement is successful the entire statement will succeed.

```
10 A = 6: B = 7: C = 8
20 IF  A = 6 AND B = 7 AND C = 7 THEN PRINT
   "AND   STATEMENT  HAS  SUCCEEDED."  ELSE
   PRINT "AND STATEMENT FAILED."
30 IF  A  = 6 OR B = 9 OR C = 10 THEN PRINT
   "OR STATEMENT HAS SUCCEEDED." ELSE PRINT
   "OR STATEMENT FAILED."
40 END
```

Try playing around with the values in line 10 to see how lines 20 and 30 work (for example, set A, B and C to equal 5, or set A to 6, B to 9, and C to 10).

Now that we've covered all of the Boolean operators – AND, XOR, NOT and OR – we should consider the effect of combining them with the use of brackets. The main consequence of introducing brackets into a statement is to allow more complicated substatements to be created. See if you can guess in advance what results will be produced by the following routine:

```
10 A = 10: B = 20: C = 30
20 IF A = 10 OR B = 10 AND B = 20 OR C = 40
   THEN  PRINT  "CALCULATION 1 WORKS." ELSE
   PRINT "CALCULATION 1 FAILED."
30 IF (A = 10 OR B = 10) AND (B = 20 OR C =
   40)  THEN  PRINT  "CALCULATION 2 WORKS."
   ELSE PRINT "CALCULATION 2 FAILED."
40 END
```

When you RUN this program you will find that it prints out:

```
CALCULATION 1 WORKS.
CALCULATION 2 FAILED.
```

But why? The answer lies in the way that the brackets have completely

changed the sense of the original statement set out in line 20. Line 20, as written, in fact consists of three separate conditions or substatements:

```
IF A = 10 OR ....
IF ... B = 10 AND B = 20 ....
IF ... OR C = 40
```

Given that the middle condition *cannot* be true (no variable can have two values at the same time within the same routine), if any one or more of the above conditions *were* 'true' then the entire statement would succeed. In other words, every one of the operators is potentially the basis for the success of the whole statement.

By adding the brackets in line 30 we have created a very different situation. In the first place there are now only *two* substatements:

```
IF (A = 10 OR B = 10) ....
IF ... (B = 20 OR C = 40)
```

Moreover, *both* statements must succeed (i.e. be 'true') in order for the whole statement to succeed. Thus either A must equal 10 *or* B must equal 10 (or both) *and* B must equal 20 *or* C must equal 40 (or both). To make this a little clearer it may help to study what are called the 'Truth Tables' for the four Boolean operators. In these tables the letter codes are as follows:

T = True
F = False
S = Successful (i.e. True)
NS = Not Successful (i.e. False)

AND

1st Condition	*2nd Condition*	*Result*
F	F	NS
F	T	NS
T	F	NS
T	T	S

XOR

1st Condition	*2nd Condition*	*Result*
F	F	NS
F	T	S
T	F	S
T	T	NS

NOT

Condition	*Result*
T	F
F	T

OR

1st Condition	*2nd Condition*	*Result*
F	F	NS
F	T	S
T	F	S
T	T	S

You can also create further Boolean operators by combining NOT with AND and OR to give NAND and NOR:

NAND

1st Condition	*2nd Condition*	*Result*
F	F	S
F	T	S
T	F	S
T	T	NS

NOR

1st Condition	*2nd Condition*	*Result*
F	F	S
F	T	NS
T	F	NS
T	T	NS

READ See DATA above.
RESTORE See DATA above.

RETURN This is the only 'legal' way of ending a section of program accessed using the GOSUB command (see above). For a detailed explanation of what happens when you leave a subroutine without using RETURN see Chapter 10.

**RIGHT$** The RIGHT$() command is used in almost exactly the same way as the LEFT$() command (see above) but this time the computer counts off characters from the right instead of from the left:

```
10 A$ = "HELLO THERE"
20 PRINT LEFT$(A$,5)
30 PRINT RIGHT$(A$,5)
```

RND The main purpose of the RND function, in adventure games at least, is to make some event or choice occur in a random fashion. On the Amstrad 464 the RND function, of itself, produces only one form of result, no matter how it is used, which is a number between 0 and 0.999999. This result can be used and influenced in several ways, however.

(a) Although the 'random number generator' within the Amstrad works quite satisfactorily without any outside assistance it can be controlled from a program. In order to make better sense of the remarks which follow you need to know just what a 'random number generator' is.

Deep within the heart of all micros (usually somewhere at the bottom of RAM) are about five bytes which hold a number which is the basis or 'seed' for any random number called up by the RND command. A number is placed in these bytes when the computer is turned on and is constantly incremented as long as the computer is in operation. Since it is extremely (and I do mean extremely) unlikely that the RND function would be used at exactly regular intervals it does produce what *appear* to be random numbers – though each number is actually taken from a constantly repeated series of numbers. This is why you will occasionally see computer-produced random numbers referred to as 'pseudo-random numbers'.

The first means of controlling the RND function is to precede it with the RANDOMIZE command. This allows us to re-set the contents of the five 'seed' bytes. If we always use the same value for 're-seeding' the generator then it will always contain the same value

when it is accessed by the RND function. In other words the random numbers become entirely *non*-random. If, on the other hand, we use the internal timer to provide the seed number – as in RANDOMIZE TIME – we will get something very near to a genuinely random result since the value of TIME itself is constantly altering.
(b) In the second case we can produce a random number within a specified range of decimal numbers (sometimes referred to as 'real numbers' or 'reals') using the form X=RND*Y. In this case we will get a value for X in the range 0 to Y-0.1 (the last value is approximate and depends on the value of Y).
(c) Next we can produce a random value within a specified range of integers (whole numbers). You will remember that I said earlier that attaching a decimal number to an integer variable usually resulted in the decimal value being rounded down. The Amstrad User's Manual says that this should happen. Yet on my machine the decimal value is actually rounded to the nearest whole number. To see what I mean run this simple test:

enter 10 a=2.3:b=2.7:a%=a:b%=b:PRINT a%,b%

According to the Manual we would expect a% *and* b% to be printed as 2. On my machine I get 2 for a%, and 3 for b%! Assuming that my machine is not an exotic rarity this means that a line in the form X%=RND*Y will produce a value in the range 0 to Y inclusive.
(d) On many occasions we will want to ensure that we get a positive result from a RND command (i.e. 1 or greater). To get a value in the range 1 to 6, for example, we would need to enter X%=RND*5+1. The first part of the equation will give a value between 0 and 5, adding 1 will raise the range by 1 at both ends – 1 to 6.

STEP This instruction is *only* used with the FOR ... NEXT statement to produce 'stepped' results rather than the normal 'one at a time' increment. For example, to divide 100 into tens, each printed on a separate line:

```
10 FOR X = 0 TO 100 STEP 10
20 FOR Y = 1 TO 10
30 PRINT X + Y;" ";
40 NEXT Y
50 PRINT
60 NEXT X
```

Note: You don't have to 'step' using whole numbers only:

```
10 FOR X = 0 TO 10
20 PRINT X
30 NEXT X
40 FOR X = 0 TO 10 STEP 2.5
50 PRINT X
60 NEXT X
```

STR$ The actual purpose of this function is to turn a numerical value (either a number or a numerical variable) into a string:

```
10 A = 10: B = 20
20 A$ = STR$(A): B$ = STR$(20)
30 PRINT A,B
40 PRINT A$,B$
```

Do note here that A$ is *not* the same as A or 10, and B$ is *not* the same as B or 20. A, 10, B and 20 can all be used as part of a mathematical calculation – A$ and B$ cannot be used as numbers except when modified using the VAL() instruction.

But why change a number into a string in the first place? Two reasons which come to mind immediately are (a) so that numerical data can be stored in a string array – to save having to create and handle a separate array, and (b) so that a large number can be read off easily one digit at a time:

```
10 A$ = STR$(857)
20 FOR X = 1 TO 3
30 PRINT MID$(A$,X,1)
40 NEXT X
```

LOCO' BASIC places a 'sign byte' at the left of a stringed value. Thus, using the example above, STR$(−857) would, of course, equal "−857". It is worth remembering, however, that the Amstrad would store STR$(857) as "(blank space)857", so that to read the digits in the number itself you would need to start at the *second* character in the string (i.e. MID$(A$,2,1), etc.).

**STRING$** While this isn't exactly the most important instruction available in LOCO' BASIC it can be very useful for manipulating a screen display. The STRING$() instruction will print out or store any repeated character without you having to type the same character over and over again. So to get a line of question marks across your screen you don't need to enter:

```
10 A$ = "?????? (40 times)"
20 PRINT A$
```

but simply:

```
10 A$ = STRING$(40,"?")
20 PRINT A$
```

which also saves about thirty-three bytes of programming space in memory! And if you only want to use the instruction once, just enter:

```
10 PRINT STRING$(40,"?")
```

TAB If you already have experience of other microcomputers you may have come across the instruction PRINT (X,Y) – known as the 'print at' command. LOCO' BASIC uses the LOCATE command to simulate this function. TAB may only be used as in:

```
10 PRINT TAB(10)"HELLO"
```

which would print HELLO starting at the 10th place (column) on the next line of the screen. The two important points to remember when using either version of TAB( ) are:

(a) The columns and lines on the screen are numbered from 1. So the leftmost column of the screen (in any MODE) is column 1, and the top line of the screen is line 1.

(b) The numbers inside the brackets in the TAB( ) instruction always refer to 'absolute' positions on the screen. So TAB(10) will move the cursor to the 10th column, and so on, no matter where it was before the instruction was executed.

THEN See IF above.

TIME The TIME variable on the Amstrad is similar to that found in some other micros. In all cases the computer's own internal 'clock' constantly tries to update the value of TIME from the moment that the computer is turned on. Thus, if we access TIME at the start of our program, we can get an exact indication of the time taken to reach any point in the program, the time taken to complete a certain routine, etc. Thus TIME is set to 0 when the machine is switched on and only the computer itself can alter the value of the variable TIME, the value being incremented by 1 every one-hundredth of a second.

A typical use for the TIME function in an adventure game would be to expand upon the INKEY and INKEY$ functions. Remember that INKEY and INKEY$ are set simply to collect a single value from the keyboard buffer. But supposing that the user presses the wrong key. That is his only chance unless we form a 'manual' loop:

```
10 T=0:A$=CHR$(INT(RND * 26) + 65)
20 X$=INKEY$:T=T+1
30 IF X$<>A$ AND T<1893 THEN 20 ELSE
   IF X$<>A$ THEN PRINT"Sorry you
   didn't press the right key." ELSE
   PRINT"Well done - you pressed the
   right key."
```

Alternatively, for a neater and easier-to-handle version of this routine, we can use the TIME function within a WHILE ... WEND loop:

```
10 T=TIME+3000:A$=CHR$(INT(RND * 26) + 65)
20 WHILE X$<>A$ AND TIME<T
30 X$=INKEY$:WEND
40 IF X$<>A$ THEN PRINT"Sorry - you
   didn't press the right key." ELSE
   PRINT"Well done - you pressed the
   right key."
```

In both versions of this routine I have aimed to give the user 10 seconds in which to match a randomly-selected capital letter which has been assigned to A$ in line 10. Although the second version may look a little longer, it is far more accurate and therefore a better piece of programming.

TO See FOR above.

WHILE ... WEND No matter what kind of micro-BASIC you may have seen before, this is one pair of commands you won't have met up with until now, which is a pity, because it really is a very useful form of the loop construct.

Since the User Manual does not deal with these commands in as much detail as it might I want to go into a little more detail than has been given to some of the other commands.

The first benefit of WHILE ... WEND loops is that they allow the condition(s) controlling the loop to be stated as it starts – you don't

have to find the end of the loop (as in REPEAT ... UNTIL) in order to find out what is going on.

If this were its only advantage then there wouldn't be much difference between WHILE ... WEND and FOR ... NEXT. In fact WHILE ... WEND has another advantage which I have already indicated – it can be controlled by not one but *any* number of conditions (within reasonable limits) linked together with the OR/AND statements. In this way, as I remarked in an earlier section, it is very easy to 'drop out' of a WHILE ... WEND loop as soon as a given situation exists without the potential problems raised by leaving a loop uncompleted.

Thirdly, there is the question of when the computer checks on the control conditions of a WHILE ... WEND loop. This may seem rather high-powered stuff for newcomers, but it is worth considering. You see both FOR ... NEXT and REPEAT ... UNTIL loops are only checked *after* each circuit of the loop has been completed – when the NEXT or UNTIL statement is reached. This means that a FOR ... NEXT/REPEAT ... UNTIL loop will always be executed at least once. A WHILE ... WEND loop, on the other hand, is checked at its start, so if the required condition has already been met then program execution will immediately pass on to the next section of the program and the loop will be ignored. I must admit that in a majority of programming situations the difference I have outlined here will not have any noticeable effect. Nevertheless, it is possible to crash a program completely by allowing a loop to operate where it is not needed, and I strongly recommend careful study of the WHILE ... WEND construct as the most effective form of loop in many, many situations.

VAL As I mentioned earlier, a number that is held in string form can only be used again as a number if handled using the VAL() function. Generally speaking, the VAL() function can be used in a variety of situations, most importantly in connection with the Object array.

In the section dealing with the AND command I showed how we might wish to check whether a player was holding certain items. This is a simple matter of checking whether or not the 'flag' (see IF) is set to −1. But if we want to find out whether a given object is in the same room as the player we have to compare the player's location (held as a *numerical* variable – PL) with the current location of the object (held as a *string* variable – OB$(CH,1)), and this we cannot do directly. What do we do, then? We simply change the string value into a numerical value and compare the two numbers:

```
10 PL = 10: OB$(CH,1) = "12"
20 IF  VAL(OB$(CH,1))  = PL THEN PRINT "TH
   OBJECT  IS  HERE."  ELSE  PRINT "THERE'
   NOTHING HERE."
30 OB$(CH,1) = "10"
40 IF  VAL(OB$(CH,1))  = PL THEN PRINT "TH
   SECOND  OBJECT  IS  HERE."  ELSE  PRIN
   "THERE'S STILL NOTHING HERE."
```

Note: Although the computer is treating OB$(CH,1) as though it were a number, the variable itself is *not* changed but remains stored as a string. So if you want to do further comparisons you will have to use the VAL() function again each and every time.

XOR This odd-looking function simply means eXclusive OR (see page 35), in other words it will only work where *one* of two conditions is true, unlike the OR function which works when either or *both* of a pair of conditions are correct. One of the most important uses of XOR is in graphics, because if you XOR a figure with its background then the background itself is not altered. The use of 'transparent' printing (see Chapter 5, page 2 of the *User Manual*) is a perfect example of XORing foreground and background colours.

Chapter Four

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 the conclusion 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 that they already know and like. The same thing applies when it comes 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 of the page will almost certainly help to generate the ideas needed 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 *Tunnels and Trolls*) 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 his 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 likely to feel that there is a tangible gap between himself and the action on the 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 of the 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 one's 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 that part of the fun of adventuring lies in the players being taken away from their everyday world of the classroom, the kitchen sink, the office or the factory floor. The stronger the link that you can create between the players and their 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 who 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, with 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 gentlemen (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 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 superhero. 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 less than a 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 short list 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 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 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 play 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 proved to be a hindrance rather than a fatality. 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, while leading or second level 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 4.1: Character status display and printout

The first program in this chapter is the *status display* I mentioned earlier. As shown here it forms a single unit with any of the character generators, though it is designed as a separate program to display the results of a given generator at any point during a game. It was first written as part of a game that was made up of a number of separate modules and when each was completed successfully – or when the hero met his doom – the game module would store the rating values in RAM, the status display program would be loaded (in place of the game module) and the *status report* would appear. That was before I discovered most of the space-saving tricks you'll find in later chapters, and the version of the program which appears here is intended as part of the master program which could be called up at any point in the game using the command STATUS, for example.

```
ERIC OLDSOCKS, OF THE DWARVES

THESE ARE YOUR CURRENT RATINGS:-

SEX: MALE                TYPE:APPRENTICE

STRENGTH: 3              HEALTH: 17

INTELLIGENCE: 11         DEXTERITY:18

HEIGHT: 6 FEET           WEIGHT: 200 LBS

WEALTH: 140 GOLD GOINS

MAX.WEIGHT YOU CAN CARRY: 30 LBS

YOUR LUCK RATING IS: 15

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

PRESS <<ENTER>> TO CONTINUE
```

Fig. 4.1. Sample of Status Display produced by Program 4.1.

```
1 REM ******  Char' Status Display  ****
*
2 :
3 :
8 REM ***  Control Routine
9 :
40 GOSUB 100:REM *** Insert chosen routi
ne at lines 100 onwards
50 GOSUB 1000
60 END
100 RETURN
997 :
998 REM ***  Check J% for Char' type
999 :
1000 IF J%<>0 THEN 1100
1027 :
1028 REM ***  If not set then set it
1029 :
1030 IF A%>10 AND C%<10 THEN J%=1:GOTO 1
100
1040 IF A%<11 AND C%>10 THEN J%=2:GOTO 1
100
1050 IF A%<8 AND C%>8 THEN J%=3:GOTO 110
0
1060 IF A%>12 AND C%>12 THEN J%=4:GOTO 1
100
1070 IF A%>7 AND C%>7 THEN J%=5:GOTO 110
0
1080 J%=6
1097 :
1098 REM ***  Get char' type from J%
1099 :
1100 ON J% GOTO 1110,1120,1130,1140,1150
,1160
1110 TY$="Warrior":GOTO 1200
1120 TY$="Wizard":GOTO 1200
1130 TY$="Thief":GOTO 1200
1140 TY$="Warwizard":GOTO 1200
1150 TY$="Delver":GOTO 1200
1160 TY$="Apprentice"
1197 :
1198 REM ***  Get kin type from I%
1199 :
1200 IF I%=0 THEN I%=1:ON I% GOTO 1210,1
220,1230,1240,1250
```

```
1210 KI$="Of the Human kin":GOTO 1300
1220 KI$="Of the Dwarves":GOTO 1300
1230 KI$="Of the Elves":GOTO 1300
1240 KI$="Of the Hobbits":GOTO 1300
1250 KI$="Leprechaun"
1297 :
1298 REM ***  Display char' status
1299 :
1300 CLS:OS=0:Q=0
1310 LOCATE 1,3
1320 PRINT #OS,NA$", "KI$:PRINT #OS:PRIN
T #OS,"These are your current ratings:-"
:PRINT #OS:PRINT #OS,"Sex: Male";TAB(22)
;"Type: ";TY$
1330 PRINT #OS:PRINT #OS,"Strength: ";A%
;TAB(22);"Health: ";B%
1340 PRINT #OS:PRINT #OS,"Intelligence:
";C%;TAB(22);"Skill: ";D%
1350 PRINT #OS:PRINT #OS,"Height: ";E%;T
AB(22);"Weight: ";F%
1360 PRINT #OS:PRINT #OS,"Wealth: ";G%;"
 gold coins"
1370 PRINT #OS:PRINT #OS,"Max. weight yo
u can carry: ";A%*3;" lbs."
1380 PRINT #OS:PRINT #OS,"Your luck rati
ng is: ";H%
1390 IF J%=6 THEN PRINT #OS:PRINT #OS,"A
s an apprentice you may choose to studyS
wordsmanship or Sorcery, but NOT both."
1400 IF Q<>0 THEN Q=0:OS=0:RETURN
1497 :
1498 REM ***  Wait for player
1499 :
1500 PRINT #OS:PRINT #OS,"Press 'P' for
Printout or 'C' to        continue: ";
1510 AN$=INKEY$:IF AN$="" THEN 1510
1520 AN$=UPPER$(AN$):IF AN$<>"P" AND AN$
<>"C" THEN 1510
1530 IF AN$="P" THEN OS=8:Q=2:GOSUB 1320
1540 RETURN
```

Program 4.1. Character status display.

The overwhelming advantage of including the status display in an 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 which is lacking in so many games currently available. Just consider for a moment the comparative effect of the display in Fig. 4.1 as against a message like:

YOU HAVE SCORED 165 POINTS

or

YOU HAVE COMPLETED 20% OF THE GAME

or

YOU NOW HAVE 350 GOLD COINS

Line-by-line analysis

Lines 40–50: Since none of the character generating programs which follow do very much by themselves, I have set up this status display program so that *any* character creating program can be slotted in to make a complete program. Thus we run the character creating routine and then the status display routine as successive procedures.

Line 1000: Not all the character generating programs go so far as to define a character type, so this line checks whether this detail has in fact been stored. If it hasn't, the player may find that his character is slightly modified on the basis of its 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 1030–1080) as the player gains or loses strength and intelligence points.

Lines 1030–1080: If used, these lines set the character type and store the relevant code as J% for future reference. You will see that the whole subroutine depends upon the values stored in A% (strength) and C% (intelligence).

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 1100–1160: Rather than try to 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 J%.

Lines 1200–1250: This section follows a similar method to lines 1100–1160, only this time we are setting the character's 'kin type'.

Lines 1300–1400: This routine is where we actually produce the status display. Each variable value is read off in turn and displayed with the relevant rating plus the conditional comment in line 1390 – see Fig. 4.1.

Lines 1500–1540: A simple method of legally 'hanging' the program until the player has had time to digest the contents of the status display. Alternatively they may obtain a printout of the STATUS DISPLAY exactly as it appears on the screen. This is achieved by simply switching the PRINT stream from 0 (the normal screen window) to 8 (the PRINTER output channel). Notice how simple this operation is. In fact we need only alter the value of OS – lines 1300 and 1530 – the computer itself does all the opening and closing of the output channel, something that most micros require the program to do (with an accompanying likelihood of a corrupted channel, especially on the Commodore 64).

Program 4.2: The built-in character

In this first, rather short, character generating 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 player may not agree that the character he has 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.

```
10  REM *****  FIXED CHARACTER  *****

20 :
30 :
100 CLS
110  LOCATE 1,8
120  INPUT "PLEASE ENTER A NAME FOR Y
     OUR CHARACTER  AND PRESS <<ENTER
     >>  ";NA$
130 A% = 10: REM ***  STRENGTH (OR ST
     AMINA)
140 B% = 10: REM ***  HEALTH
150 C% = 10: REM ***  INTELLIGENCE
160 D% = 10: REM ***  SKILL
170 E% = 6: REM ***  HEIGHT IN FEET
180 F% = 180: REM ***  WEIGHT IN POUN
     DS
190 G% = 1: REM ***  WEALTH UNITS / 1
     0
200 H% = 10: REM ***  LUCK
210  RETURN
```

Program 4.2. Fixed character.

Line-by-line analysis

Line 100: Here we simply define the character generator as a procedure, remembering that *none* of the four generators will do very much unless used together with the character display program (Program 4.1). This can be done using CHAIN MERGE.

Lines 110–120: Having cleared the screen and moved the cursor down to the start of the eighth line, a simple display asks for the player's (fantasy) name which is then stored as NA$.

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

Fig. 4.2. Screen display from Program 4.2, line 120.

Lines 130–200: In all versions of the character generating program we will be dealing with eight basic characteristics and later versions add a 'character type'. The display routine adds one more, the 'kin type'. The first eight characteristics are set out in lines 130–200 and the display routine will automatically handle the other two. All characteristics are stored as *integer* variables which will be lost if another program is CHAINed or CHAIN MERGEd once the character generator/display has been RUN.

Program 4.3: Off-the-peg characters

Our third program is a direct offshoot of Program 4.2. This time the player has the option of choosing one character type from a preset list. As in Program 4.2, however, the ratings for each character are still fixed by the writer. The advantages here are the same as before and 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 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.

```
10  REM ***  FIXED CHARACTERS WITH SE
    LECTION  ***
20 :
30 :
98  REM ***** SET UP CHARACTER ARRAY
99 :
100  DIM CT$(4)
110  FOR X = 1 TO 4
120  READ CT$(X)
130  NEXT
197 :
198  REM *****  PLAYER CHOOSES NAME A
    ND CHARACTER TYPE
199 :
200 CLS: LOCATE 1,8
210  INPUT "PLEASE ENTER A NAME FOR Y
    OUR CHARACTER  AND PRESS <<ENTER
    >>  ";NA$
220 CLS: LOCATE 1,4 : PRINT NA$;" PLE
    ASE SELECT YOUR": PRINT
230  PRINT "CHARACTER TYPE FROM THE L
    IST BELOW:": PRINT
240  FOR VT = 1 TO 4
250 Q = VT * 2 + 6
260  LOCATE 1,Q : PRINT "(";VT;") ";C
    T$(VT)
270  NEXT
280  LOCATE 1,18 : PRINT "ENTER NUMBE
    R OF CHARACTER TYPE NOW ";
```

```
290 CN$ = INKEY$: IF CN$ = "" THEN 29
    0
300 CN =  VAL (CN$): IF CN < 1 OR CN >
    4 THEN  LOCATE 1,20 : PRINT "YOU
     MUST CHOOSE A NUMBER BETWEEN 1
    AND 4": GOTO 290
397 :
398  REM *****  COLLECT RIGHT SET OF
    RATINGS
399 :
400  FOR X = 1 TO 4
410  READ A%,B%,C%,D%,E%,F%,G%,H%
420  IF X = CN THEN  GOSUB 600:X = 4
430  NEXT X
440 J% = CN: IF J% = 4 THEN J% = 5
450  RETURN
497 :
498  REM *****  DETAILS OF CHARACTER
    RATINGS
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
597 :
598  REM ***  CLEAR UNUSED RATINGS DA
    TA
599 :
600  FOR CL = X + 1 TO 4
610  READ A,B,C,D,E,F,G,H
620  NEXT CL
630  RETURN
```

Program 4.3. Fixed characters with user-selection.

Line-by-line analysis

Lines 100–130: These lines set up, or 'initialise', a 4×1 array to hold the four character types named in line 500. (If you aren't familiar with arrays and their uses, see Chapter 5 for more details.) We then use a simple FOR ... NEXT loop to store the names in line 500 in the array CT$().

Lines 200–210: These lines clear the text screen and collect a name for the player's character which will be stored as NA$.

```
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. 4.3. Screen display from Program 4.3, lines 220–280.

Lines 220–230: The printout is the heading for a simple screen display, or 'menu', which allows the player to choose his character type from a preset list.

Lines 240–270: 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 280–290: These lines finish off the menu and include 'error trap' so that only the legal numbers – 1, 2, 3 and 4 – will be accepted by the computer.

Lines 400–430: Here we have a loop structure which will collect from one to four sets of character ratings. It is short-circuited in line 420, if necessary, by resetting X to the last value in line 400 (in this instance, 4). We thus avoid jumping out of the loop before it is complete – something generally regarded as a 'bad habit'. See Chapter 10 for more details. Eight character ratings are actually collected from the DATA in lines 510–540 and stored in the variables A% through to H%, ending when the required set is in store.

Lines 440–450: Since we have given the player a choice of character, we can store a value for that too – in J%. We then RETURN to the program head to move on to the character display.

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

Lines 510–540: Initial storage space for the character ratings according to character type.

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 4.2 and 4.3 above show the actual screen displays generated by Program 4.3. The word BEOWULF, in Fig. 4.2, was of course input from the keyboard.

Program 4.4: User-controlled character type and ratings

The third program of this series, though it starts out looking a lot like Program 4.3, 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 which 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.

```
10 REM *****  User-set Characters  *****
20 :
30 :
98 REM ***  Set up character array
99 :
100 DIM CT$(4),H1(4),H2(4),CT(8)
110 FOR X=1 TO 4
120 READ CT$(X),H1(X),H2(X)
```

```
130 NEXT
197 :
198 REM *** Player chooses name and cha
racter type
199 :
200 CLS:LOCATE 1,8
210 INPUT "Please enter a name for your
character and press <<ENTER>> ";NA$
220 CLS:LOCATE 1,4:PRINT NA$;" Please se
lect your":PRINT
230 PRINT "character type from the list
below:":PRINT
240 FOR VT=1 TO 4
250 Q=VT*2+6
260 LOCATE 1,Q:PRINT "(";VT;") ";CT$(VT)
270 NEXT
280 LOCATE 1,18:PRINT "Enter number of c
haracter type now ";
290 CN$=INKEY$:IF CN$="" THEN 290
300 CN=VAL(CN$):IF CN<1 OR CN>4 THEN LOC
ATE 1,20:PRINT "You MUST choose a number
 between 1 and 4":GOTO 290
397 :
398 REM *** Player chooses ratings
399 :
400 CLS:LOCATE 1,3:PRINT NA$;" you have
48 rating":PRINT:PRINT "points to be div
ided between 6 characterqualities (using
 whole numbers only!)":PRINT
410 PRINT "Each quality must be given at
 least 1   point.  No quality may be giv
en more    than 12 points!":PRINT
420 LOCATE 1,12:PRINT "(1) Strength";TAB
(22);"(4) Skill":PRINT
430 PRINT "(2) Health";TAB(22);"(5) Weal
th units/10"
440 PRINT "(3) Intelligence";TAB(22);"(6
) Luck":PRINT
450 FOR X=1 TO 6
460 LOCATE 1,18:PRINT "Enter points for"
X;:INPUT "and press <<ENTER>> ";CT$
```

```
470 CT(X)=VAL(CT$):IF CT(X)<1 OR CT(X)>1
2 OR CT(X)<>INT(CT(X)) THEN LOCATE 1,20:
PRINT CHR$(18);"ILLEGAL ENTRY!!!":GOTO 4
60
480 IF CH+CT(X)+(6-X)>48 THEN LOCATE 1,2
0:PRINT CHR$(18);"You only have";48-CH;"
points left!!!":GOTO 460
490 CH=CH+CT(X):LOCATE 1,20:PRINT CHR$(1
8):LOCATE 1,18:PRINT CHR$(18)
497 :
498 REM ***  Details of character rating
s
499 :
500 NEXT X
506 :
507 REM ***  Transfer ratings and
508 REM      erase redundant arrays
509 :
520 A%=CT(1):B%=CT(2):C%=CT(3):D%=CT(4)
530 E%=H1(CN):F%=H2(CN):G%=CT(5):H%=CT(6
)
540 J%=CN
550 ERASE CT$,CT,H1,H2
560 RETURN
597 :
598 REM ***  Character types, heights an
d weights
599 :
600 DATA Warrior,6,200
610 DATA Wizard,6,150
620 DATA Thief,5,140
630 DATA Delver,5,160
```

Program 4.4. User-set characters.

Line-by-line analysis

Lines 100–130: We start, this time, by creating four arrays, the first three of which are filled from the DATA in lines 600–630 using the simple loop in lines 110–130.

Lines 200–300: This is almost a direct copy of lines 200–290 of Program 4.3. See the corresponding line notes for details.

Lines 400–440: These statements set up the whole of the screen display shown in Fig. 4.4 except the last line.

```
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 <<ENTER>>
```

Fig. 4.4. Screen display from Program 4.4, lines 400–460.

Line 450: Sets up a standard FOR...NEXT loop which will be executed six times.

Line 460: Produces the last line in Fig. 4.4. Note that the value of X actually prints out as part of the query.

Line 470: 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 and line 460 is repeated.

Note: When line 460 is repeated (after lines 480 or 490) 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 for the sake 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 480: 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 the last input is rejected. The player is then told how many rating points he has left and the program returns to line 460.

Lines 490–500: If the input is satisfactory in all respects it is added to CH – the total number of rating points used to date. Next we print CHR$(18), to erase any earlier message, and the query line is blanked out entirely – including the last input. Then, if necessary, the program returns to the start of the loop.

Lines 520–560: 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 the CT( ) array into integer variables, rearranging the values in CT(5) and CT(6) – Wealth and Luck – and the Height and Weight values in H1(CN) and H2(CN) accordingly, and all the arrays can be wiped out.

Lines 600–630: 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 of the DATA in lines 600–630 is collected at the start of this program I do not need the 'clear out' subroutine used in Program 4.3.

Program 4.5: 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 120 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 and thin as a pole – 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 player-controlled characters? In the first place this program allows an even wider range of characters than did Program 4.4, enhancing the challenge factor of the game. Secondly, the player is

still allowed a measure of control over the character. This routine would appear at the start of an adventure, and if the player didn't like the character that the computer had generated he could 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.

```
1  REM *****  COMPUTER-SET CHARS'  **
     ***
2 :
3 :
98  REM *** SET UP CT() ARRAY
99 :
100  DIM CT(8)
110  FOR X = 1 TO 8
120 CT(X) =  INT ( RND  * 15) + 3
130  NEXT X
150  FOR X = 1 TO 8: PRINT CT(X): NEXT

163 :
164 :
165  REM *** MODIFICATION ROUTINES
166 :
167 :
168  REM *** RE-SET KIN TYPE
169 :
170 I% =  RND  * 4 + 1
297 :
298  REM *** RE-SET HEIGHT & WEIGHT
299 :
300 E% = CT(5)
310  IF E% < 4 THEN E% = 4:F% = 110: GOTO
     400
320  IF E% < 7 THEN E% = 5:F% = 130: GOTO
     400
330  IF E% < 10 THEN E% = 5:F% = 160:
      GOTO 400
340  IF E% < 13 THEN E% = 6:F% = 190:
      GOTO 400
350  IF E% < 16 THEN E% = 6:F% = 220:
      GOTO 400
360 E% = 7:F% = 250
396 :
397  REM *** MODIFY RATINGS ACCORDING
```

```
398  REM          TO KIN TYPE
399 :
400 A = CT(1):B = CT(2):C = CT(3):D =
    CT(4):E = E%:F = F%:H = CT(8)
410  IF I% = 2 THEN A = A * 2:B = B *
    2:E = E * .7:F = F * .8
420  IF I% = 3 THEN B = B * .7:C = C *
    1.5:D = D * 1.5:E = E * 1.25
430  IF I% = 4 THEN A = A / 2:B = B *
    2:D = D * 1.5:E = E / 2:F = F /
    2
440  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
450 CT(1) = A:CT(2) = B:CT(3) = C:CT(
    4) = D:CT(5) = E:CT(6) = F:CT(8)
     = H
496 :
497  REM *** RE-SET CT(1) - CT(8)
498  REM         AS INTEGERS
499 :
500  FOR X = 1 TO 8
510  CT(X) = CINT(CT(X))
520  NEXT X
536 :
537  REM *** RE-ASSIGN CT() ARRAY AND

538  REM           THEN ERASE IT
539 :
540 A% = CT(1):B% = CT(2):C% = CT(3):
    D% = CT(4)
550 E% = CT(5):F% = CT(6):G% = CT(7):
    H% = CT(8)
560  ERASE CT
597 :
598  REM *** GET NAME FOR CHAR'
599 :
600 CLS: LOCATE 1,6
610  INPUT "PLEASE ENTER A NAME FOR Y
    OUR CHARACTER  AND PRESS <<ENTER
    >>  ";NA$
620  RETURN
```

Program 4.5. Computer-set characters.

Line-by-line analysis

Lines 100–130: First we set up a single array – CT() with eight elements (plus the 'zero element'). This is then filled with eight 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 150: 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.

Line 170: Since I%, the value for the *kin type* – Human, Dwarf, etc. – will not undergo modification in the following routine, I have set it immediately to a random value between 1 and 5.

Lines 300–360: 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 integer values have been set rather than go on 'falling through' the rest of the IF statements.

Lines 400–450: Here all relevant character qualities are modified to fit particular kin types. If I%=1 then the kin type is Human and the character keeps his given rating. 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 440).

Lines 500–560: Another FOR...NEXT loop which turns all CT() values except CT(9) – which is already an integer – into whole numbers. It does this by rounding them to the *nearest* whole number, not by rounding them down as would happen if we transferred the values straight to the integer variables. Again the temporary array is ERASEd.

Note: In many machines, including the Amstrad, 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 line 510 by using CINT().

Line 600–610: The familiar routine for getting the player's (fantasy) name as NA$. But remember, if you CHAIN another program on *after* using this or any other character generator, the values and the integer variables assigned to NA$ will be lost and must be collected again.

Program 4.6: 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 and a second or third level opponent – in this case George and a dragon. It also serves to show how easily the character ratings, once set, can be altered from within a program. If you are using a routine like this in a program which includes a character generator then the variables in line 10 will already exist, at least for the player.

By the way, you might like to notice how the player's LUCK rating is being used here. The player's *combat* strength is calculated by adding his actual strength (CT(1) or A%) to his luck rating (CT(8) or H%) 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 its 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 routine over two hundred times I found that the dragon was winning only five percent of the battles! I therefore raised the dragon's strength rating 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 might choose to omit these two lines and simply display the result.

```
1  REM  ***  A SIMPLE COMBAT ROUTINE
       ***
2 :
3 :
10 SL = 5:SH = 12:LK = 3:PT = SH + LK
     :ML = 2:MH = 26
20 PS% = (SL * ( RND  * 6 + 1) + PT) /
     6
30 MH = MH - PS%: IF MH < 1 THEN 100
35  PRINT "DRAGON = "MH
40 MS% = (ML * ( RND  * 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    DRAGON!":
      END
150  PRINT : PRINT "WHOOPS! DRAGONS 1
     , YOU'VE LOST.  R.I.P. ": END
```

Program 4.6. George and the Dragon (a combat routine).

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 it can actually *kill* the player's character), doesn't 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 the effect of the Player'S hit and is calculated by the same means as the Monster'S hit in line 40. Thus the effect of character blows is found by multiplying his skill rating by a random number between 1 and 7 (as if we were throwing a dice). We then add on the character's *current* strength (i.e. the player's combat strength rating), divide the total by 6 to make the combat last a bit longer, and finally round the result down by assigning it to an integer variable.

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 below 1. 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 variable names 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 his strength rating will be very low at this point – over four hundred combat sessions his average strength at line 100 was 2! This is something that you must allow 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 make sure that it is the player's *strength* rating that is affected by the combat and not just his *combat* strength rating.

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 upon 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 have already seen how a game program can be split up into separate subroutines or 'modules'. This approach to game design – known as 'top down' design – will be discussed at greater length in Chapter 10. But for the time being I would just say that the best way I know of getting to understand how a program or subroutine works and how you can perhaps improve on it – is by pulling it to pieces and altering it in small ways here and there. Try leaving out a line or two, and see what difference it makes. Try altering the values of some of the variables. And if you can't follow what is happening to some of the variables add in a few extra lines to get a printout of their values from time to time. (I'll give a couple of routines which do this job in Chapter 6.) The only cardinal rule you should *never* break is this: don't experiment with any program – including your own – until you have one or two copies stored safely on tape or disk which cannot come to any harm if the experiment goes wrong!

Chapter Five

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 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 thought of a storyline for your adventure then choosing the landscape within which the action will take place should come fairly easily. Some readers, on the other hand, may find it easier to start with a setting for their adventure and developing 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.

Firstly, 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 just one general location – a country house, on board a space ship, or wherever – as are *Cranston Manor*, *Star Cross* and several other successful games. In *Deadline*, for example, all of the action takes place in and around 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 the 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. Compare Fig. 5.1 with Fig. 5.2 as examples of what can be achieved in this way.

(2) The act of travelling can become even more interesting if the 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 a 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 6 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 rewritten so that

---

You are outside the green house. A car is parked at the kerb with the ignition key still in the lock.

WHAT NOW? GO SOUTH

After walking for an hour you reach the edge of town. Mile after mile of empty desert stretches in front of you. This is a good time to be driving a car!

WHAT NOW? GO NORTH

Fig. 5.1. Two 'rooms' in an adventure game.

You are outside the green house. A car is parked at the kerb with the ignition key still in the lock.

WHAT NOW? ENTER THE CAR

You're in the car.

WHAT NOW? DRIVE SOUTH

You can't do that yet.

WHAT NOW? START ENGINE. DRIVE SOUTH

You drive to the edge of town and out into the desert. After driving 100 miles you come to a sun-bleached wood cabin. It seems to be a petrol station.

WHAT NOW? EXAMINE FUEL GAUGE

You're nearly out of petrol.

Fig. 5.2. From the same adventure – see how a different command (*DRIVE* SOUTH) produces an entirely different result.

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 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 primary 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 7 and 8. 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 in very finite amounts. So the next step is to decide roughly how big your adventure can be.

The first thing we need to remember when planning the overall size of a program is that it has to be stored on tape. Anything stored on tape can only be written to and collected from the tape using 'sequential' files. This 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. The process of reading from or writing to a tape file is also extremely slow. (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.) This 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. This means that our adventures must be written in one of two ways. Either they will have to be fairly small in order that we can fit a complete adventure into the computer at one go, or they will have to

be written in such a way that one larger game is made up of several smaller parts (or *modules*) which can be LOADed one after another to make up the complete adventure.

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.

This leads on to another very important question: how much 'user RAM' do you have in your machine (in other words, how much room is allowed for ordinary BASIC programs)?

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 which are 16K machines. The BBC B and the Electron are 32K machines. The Oric Atmos, the larger Spectrum, the Atari 800 and the unmodified Apple II series are all listed at 48K. Finally, the Commodore 64 and the Amstrad CPC464 weigh in at 64K! These figures will already be familiar to most micro owners, I know. The reason why 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 Electron – from 8.5K to 20.5K (maximum)
The BBC B – 27–28K
The Oric Atmos (without high resolution graphics) – about 46K
The Spectrum 48K – about 40K
The Atari 800 – about 37.5K
The Apple II+ (minus high resolution graphics *and* DOS) – 50K
The Commodore 64 – a few bytes less than 38K
The Amstrad CPC464 – 42.5K

In point of fact the C64 and the Amstrad actually have more than 64K, and all 48K micros are actually 64K machines! In other words they

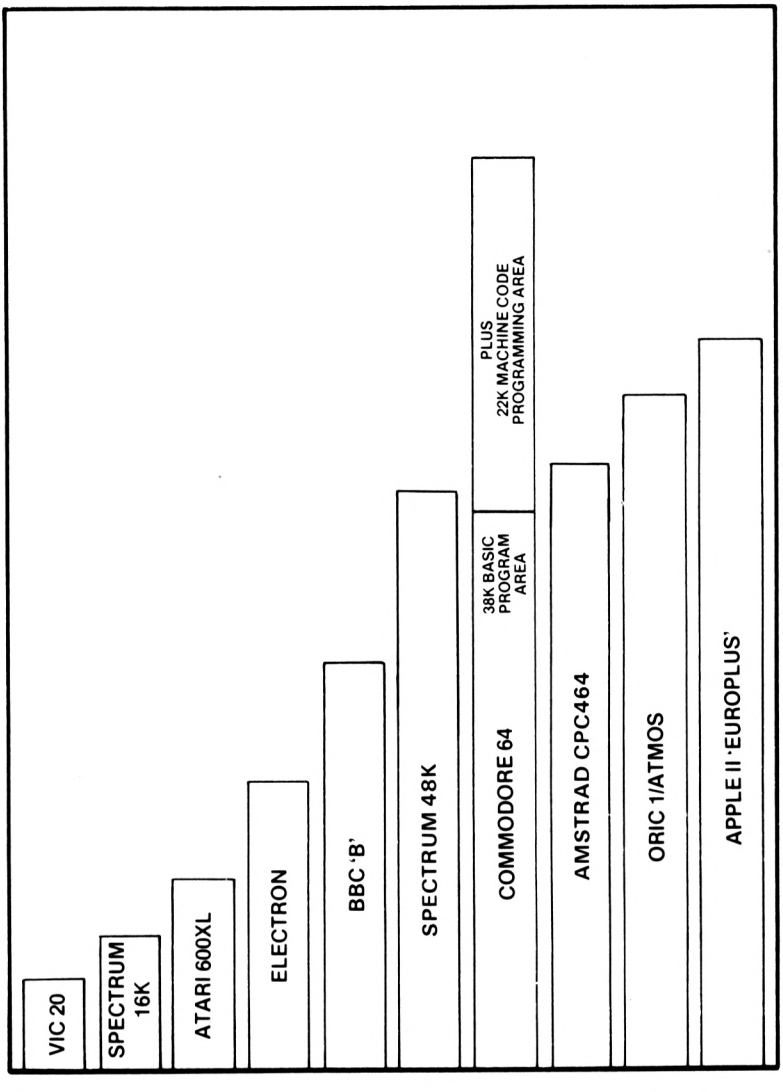


Fig. 5.3. Chart showing comparative size of 'user' RAM space in ten best-selling micros.

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 (using

machine code). Unfortunately this is only the first step in calculating the total space needed. We must also allow storage space 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 7). 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 11 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 50 locations for every 16K of user RAM. For the Amstrad that comes to about 110–120 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 just 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; and 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 highly 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.

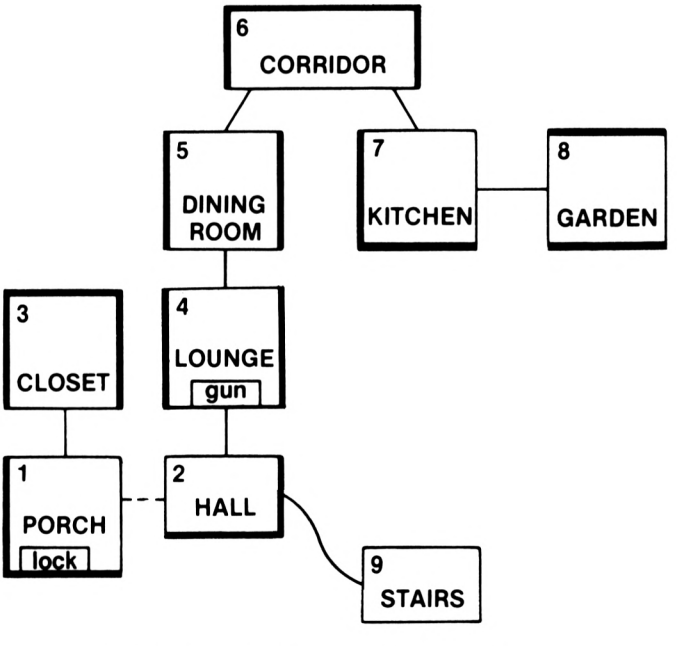


Fig. 5.4. Section of a typical boxes and lines map.

The most positive thing that can be said 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. 5.4. 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 the 'consistency rule'. This can be seen quite clearly in Fig. 5.4 in boxes 5, 6 and 7. In this area the player can move into room 6, a corridor, from either room 5 or 7 by using the command GO NORTH. But whatever 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, 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 OR RIGHT DOOR? But unless this choice will appear several times, the programmer has wasted space dealing with a situation which 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 that 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.

Program 5.1: 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 x B grid below (Fig. 5.5) 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 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.

```
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 5.1. Calculated moves in a 10 × 10 grid.

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. 5.5. Linked squares map frame (A × B grid). If an 'unmodified' calculated move routine were used together with this map then off-the-edge errors could occur when the player is in any of the thirty-six squares *outside* the thick inner boundary line.

Program 5.2: A 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 can be seen from the second version of the program (Program 5.2), but again this is a system which uses up valuable space to no good purpose. Indeed, as will be seen in Chapter 7, the use of *calculated* movements is a waste of time. Thus the linked squares map in Fig. 5.6 is just as satisfactory as, and rather less restricted than, the symmetrical layout in Fig. 5.5.

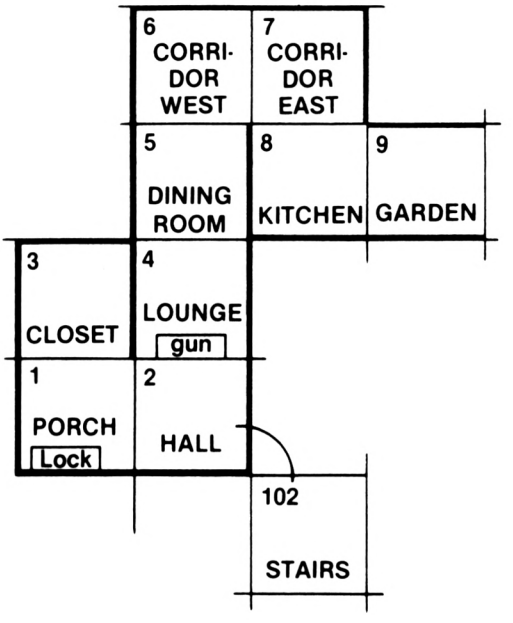


Fig. 5.6. Opening section of a linked squares map.

```
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 : PRINT "YOU CANNOT MOVE I
     N THAT DIRECTION"
310  RETURN
```

Program 5.2. Modified calculated moves in 10 × 10 grid.

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. Having said that, I must emphasise the fact that maps of this kind take up a great deal of space – and involve quite a lot of extra preparation. Using linked octagons does not increase the size of the room description file, but it will make your movement codes significantly larger. Although the Amstrad has quite a large amount of RAM space – about 42K when using tapes – you might find it easier to stick to linked squares when you plan your first adventure.

The main advantage to 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.

... Tuppence 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 the aid of 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 redraw 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 a place 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 7.

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. 5.7 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

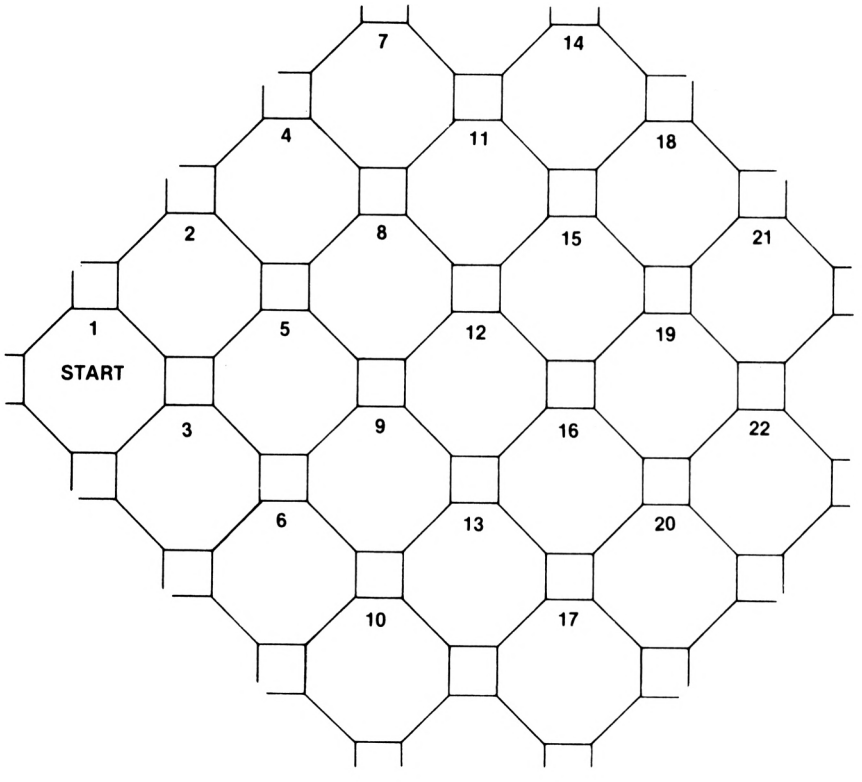


Fig. 5.7. Basic linked octagons map.

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 straight lines – paths 1, 2, 4,7,etc.or 1, 3, 6, 10. But if the adventure started in a forest or an underground 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 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 easier to follow when you start programming.

Colour coding

If you turn to Fig. 5.8 you'll see that it has altered quite significantly from the simple framework in Fig. 5.7. 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.

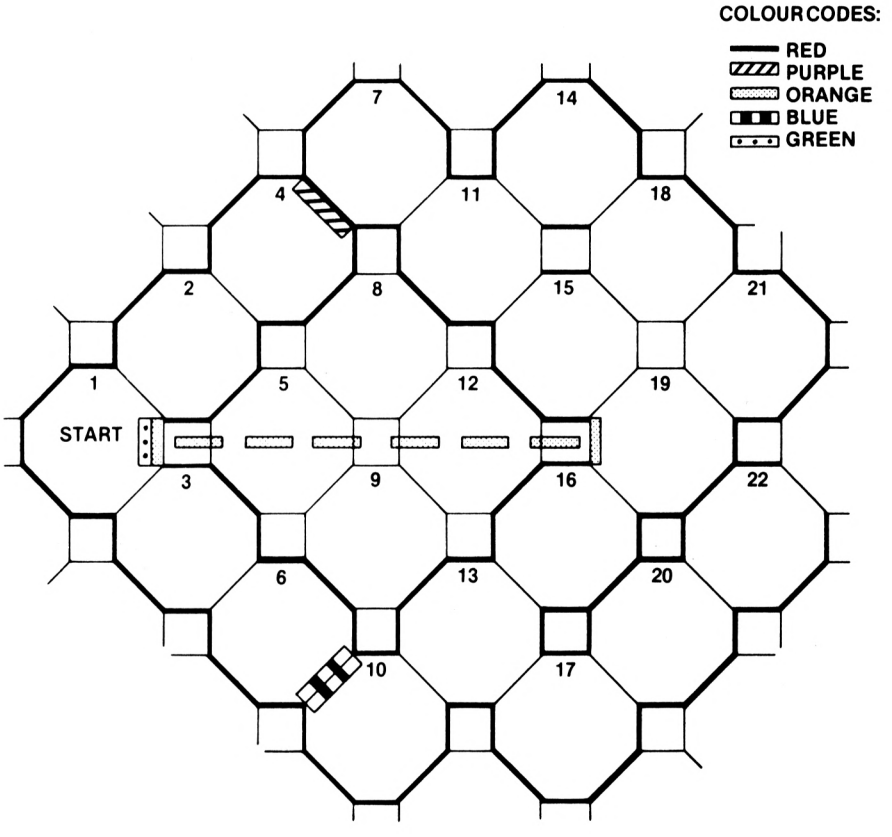


Fig. 5.8. Linked octagons map with 'coloured' details.

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 codes 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 which 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 in again – if he dives from a cliff edge into a river, say. In this situation the *purple* line marks the boundary of the room he may leave, while a *red* line marks the boundary of the room he 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 in a spaceship 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, then, 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 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 which 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 boundary line on one side of the boundary to indicate that the player may not move in that direction without satisfying some previous condition. If you look at Fig. 5.6 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 northeastern 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. When 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 7.)

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 which 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 show the difference between the two situations on the map. Blue lines will usually mean 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*). An example of this can be seen at the entrance to the tunnel in room 1 on my map. At this end of the tunnel the entrance – a trap door, perhaps – is hidden from view, though the entrance at the other end (in room 19) is plainly visible.

And now, having colour coded all the sections of your map, you're nearly ready to start programming your adventure.

Chapter Six

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 4, 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.

Elementary, my dear Arnold

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 functions 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$(3), etc. In Fig. 6.1 you will see that it holds a list of five names. So in order to create the A$() array prior to its use we would use the statement:

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

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 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. 6.2). Some machines – the Lynx, for instance – only allow one-dimensional arrays, while others allow truly labyrinthine arrays with up to 88

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

Fig. 6.1. Example of a one-dimensional array for A$( ).

			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. 6.2. Example of a two dimensional array for A$( ). Notice the number of 'zero' (UNUSED) elements.

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 Amstrad, start arrays from 0. Just how big a difference the zero elements make can be seen in the two diagrams above. And remember that the computer sets aside RAM space, *with pointers*, for *all* the elements in a DIMmed array – not just the elements that are actually in use. Thus each element takes up at least 3 bytes of memory even when it's empty. So don't waste space unnecessarily by creating a separate array for every list if you can possibly use the same array for several different routines.

Since the Amstrad does assign 0 (zero) as the lowest value in all arrays, 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)
```

But 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, why not treat it as a separate one-dimensional array?

Program 6.1: 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.

```
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 6.1. Array demonstration.

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 4 I spent a good deal of time explaining how to create 'progressive' characters – characters which would be modified by their experience in the course of an adventure. On the assumption that many writers will appreciate the value of the progressive factor and will 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 4 – 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 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 having 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!

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 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 6.2: Picking up the pieces

This first routine creates a simple array containing eight objects which

the player can GET. It illustrates both the GET command and I, or INVENTORY, but does *not* take account of the player's STRENGTH rating or the weight of the objects.

```
1  REM *****  GET OBJECT #1  *****
2 :
3 :
8  REM *** SET UP DEMO
9 :
10 CLS: GOSUB 1000
20 T = 0
30  PRINT : PRINT : INPUT "WHICH ROOM
    ";PL$
40 PL =  VAL (PL$): IF PL < 1 OR PL >
    8 THEN 30
97 :
98  REM *** 'PARSE' COMMAND INPUT
99 :
100  PRINT : PRINT : INPUT "WHAT NOW
    ";CO$
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$,1) = "Q" THEN  GOTO
    900
140  PRINT : PRINT "SORRY - I DON'T U
    NDERSTAND ";CO$: 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$(X,1) THEN CH = X: GOTO
     600
520  NEXT
530  PRINT : PRINT "I SEE NO ";N$;" H
     ERE."
540  RETURN
597 :
598  REM *** TRY TO GET OBJECT
599 :
600  IF IN > 5 THEN  PRINT : PRINT "S
     ORRY - YOU CAN ONLY CARRY SIX IT
     EMS.": RETURN
610  IF OB$(CH,2) = "-1" THEN  PRINT
     : PRINT "YOU ALREADY HAVE THE ";
     N$;"!": RETURN
620  IF OB$(CH,2) = "0" THEN  PRINT :
      PRINT "SORRY - ";N$;" ISN'T AVA
     ILABLE!": RETURN
630  IF  VAL (OB$(CH,2)) <  > PL THEN
      PRINT : PRINT "THE ";N$;" ISN'T
      HERE!": RETURN
640  PRINT : PRINT "O.K. - YOU HAVE T
     HE ";N$
650 OB$(CH,2) = "-1":IN = IN + 1: RETURN

897 :
898  REM *** QUIT ROUTINE
899 :
900 CLS: 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 6.2. Get object (depending on quantity).

Line-by-line-analysis

Line 10: Having cleared the screen, program execution passes to lines 1000–1030 which fill 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 PL – 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. 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 six 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 line 640 tells you if you have successfully obtained the required objects. 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: Like I said – Q quits.

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

Program 6.3: How to solve a 'weighty' problem

In our second inventory routine we create a more realistic situation by using the MAX. WEIGHT YOU CAN CARRY condition given in the status display program in Chapter 4 (see Program 4.1). If the player tries to pick up an item which will take the total inventory weight over the maximum allowed then the message SORRY – YOU CAN'T CARRY ANYTHING THAT BIG is displayed and the command is rejected.

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.

```
1  REM *****  GET OBJECT #2  *****
2 :
3 :
8  REM *** SET UP DEMO
9 :
10 CLS: GOSUB 1000
20 T = 0
30  PRINT : PRINT : INPUT "WHICH ROOM
      ";PL$
```

```
40 PL =  VAL (PL$): IF PL < 1 OR PL >
    8 THEN 30
97 :
98  REM *** 'PARSE' COMMAND INPUT
99 :
100  PRINT : PRINT : INPUT "WHAT NOW
    ";CO$
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$,1) = "Q" THEN  GOTO
    900
140  PRINT : PRINT "SORRY - I DON'T U
    NDERSTAND ";CO$: GOTO 20
297 :
298  REM *** DISPLAY INVENTORY
299 :
300  PRINT : PRINT "YOU ARE CARRYING:
     "
310  FOR X = 1 TO 8
320  IF OB$(X,1) = "-1" THEN  PRINT "
    THE ";OB$(X,0)
330  NEXT
340  PRINT : PRINT "A TOTAL WEIGHT OF
     ";IN;" LBS."
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: GOTO
    600
520  NEXT
```

```
530  PRINT : PRINT "I SEE NO ";N$;" H
     ERE."
540  RETURN
597 :
598  REM *** TRY TO GET OBJECT
599 :
600  IF IN +  VAL (OB$(CH,2)) > 20 THEN
      PRINT : PRINT "SORRY - YOU CAN'
     T CARRY ANYTHING THAT BIG.": RETURN

610  IF OB$(CH,1) = "-1" THEN  PRINT
     : PRINT "YOU ALREADY HAVE THE ";
     N$;"!": RETURN
620  IF OB$(CH,1) = "0" THEN  PRINT :
      PRINT "SORRY - ";N$;" ISN'T AVA
     ILABLE!": RETURN
630  IF  VAL (OB$(CH,1)) <  > PL THEN
      PRINT : PRINT "THE ";N$;" ISN'T
      HERE!": RETURN
640  PRINT : PRINT "O.K. - YOU HAVE T
     HE ";N$
650 OB$(CH,1) = "-1":IN = IN +  VAL (
     OB$(CH,2)): RETURN
897 :
898  REM *** QUIT ROUTINE
899 :
900 CLS: 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 6.3. Get object (depending on weight).

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 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 6.4: 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 using the command DROP (item), and in this example I've stuck to that simple format.

```
1 REM *****  Drop object  *****
2 :
3 :
6 REM ***  This routine assumes that
7 REM           CO$="DROP THE KNIFE"
8 REM       and N$(NP)="THE KNIFE"
9 :
10 FOR X=1 TO F1
20 IF OB$(X,0)=N$(NP) THEN TE=X:GOTO 100
30 NEXT
40 PRINT:PRINT "Sorry - can't find ";N$(
NP):RETURN
```

```
97 :
98 REM ***  If N$(NP) exists check locat
ion
99 :
100 IF OB$(TE,1)<>"-1" THEN PRINT:PRINT
"You can't drop what you don't have!!":R
ETURN
110 PRINT:PRINT "O.K.":OB$(TE,1)=STR$(PL
):RETURN
```

Program 6.4. Drop object.

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, we politely tell the player what an idiot he is and go back for another command.

Lines 100–110: In line 110 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 110, either IN=IN–1 or IN=IN–VAL (OB$(TE,2)).

Strength is not enough ...

Health

The health character rating keeps track of the player's physical condition. Thus 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 whilst 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 worry 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 feel that it is worth ignoring 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 4 I use intelligence as one of the guidelines for deciding what *type* of character the player was given. I assume 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, while 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 broad swords), 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 would a 110 lb midget.

If you are short of space then height and weight, like health, are usually the 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 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 in to 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 trapdoor 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 option 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 reason why overcrowding usually occurs – through 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 clever use of machine code? In a 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 the programs will RUN by themselves. To see them at work they must be accompanied by a command input routine (to move the player's character), a set of movement codes (see Chapter 7), and a means of checking what is happening to the Random Item (or Items) such as a simple PRINT RI (RI being the variable which holds the Random Item's location) after each move. A full, working version of the Random Item/Chase routine can be found in the game listing in the Appendix.

Program 6.5: 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 tries to move 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.

```
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%=RND*PD+1:Q%=Q%+RI*PD
3010 NM=PEEK(BA+Q%):IF NM=0 THEN RETURN
3016 :
3017 REM ***  If NM valid try to move RI
3018 REM      towards player and 'return'

3019 :
3020 IF NM=PL THEN RI=NM:GOTO XXX
3030 IF PL<RI AND NM<RI THEN RI=NM:RETUR
N
```

```
3040 IF PL>RI AND NM>RI THEN RI=NM:RETUR
N
3047 :
3048 REM ***  Else 'return' anyway
3049 :
3050 RETURN
```

Program 6.5. Random item chases player.

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 Program 6.7.

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 the expression 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 for 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 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 reset 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. For those readers who have never used any programming language except BBC BASIC I should perhaps apologise for any confusion caused by the references to PEEK and POKE. Although these *operations* are included in BBC BASIC they are represented by symbols rather than words. If you are not already familiar with these operations, by any name, you will find a complete explanation of the principles involved in the next chapter.

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% = RND*1
3010 IF Q% = 0 THEN NM = RI + 1
3020 IF Q% = 1 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, 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 up with the item since he has no way of knowing where it is – unless you deal with that elsewhere in your program.

Program 6.6: The watchdog

In our second routine 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.

```
1 REM *****  RI/Limited move  *****
2 :
3 :
8 REM ***  Place RI (RI area=40 to 45)
9 :
10 RI=43
301 ' :
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%=RND*PD+1:Q%=Q%+RI*PD
3006 :
3007 REM ***  Check result against
3008 REM       the movement codes
3009 :
3010 NM=PEEK(BA+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 6.6. Random item with limited movement.

Line-by-line analysis

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 previous program.

Line 3030: If changing RI to the value of NM would take the item outside its allotted 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 6.7: The random monster

In our 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.

```
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%=RND*TR+LR
27 :
28 REM ***  Filter out repeats
29 :
30 FOR Y=1 TO X-1
40 IF VAL(OB$(Y,1))=T% THEN Y=X-1:FL=1
50 NEXT Y:IF FL=1 THEN FL=0:GOTO 20
57 :
58 REM ***  And place RI
59 :
60 OB$(X,1)=STR$(T%)
67 :
68 REM ***  Then repeat for next RI
69 :
70 NEXT X
80 END
```

Program 6.7. Routine to place random item(s).

Line-by-line analysis

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

Line 20: Get 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 simplfying 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.

Too many cooks ...

The last objection to the use of 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, you will have to take into account the fact 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 the objects and problems interrelated rather than by 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 be used as a weapon, and since this will inevitably lead to its 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 6.8: 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.

```
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)
     ,(OB1 WEIGHT),(OB2 NAME),(OB2 LO
     CATION),(OB2 WEIGHT)
10010  DATA ETC.
```

Program 6.8. Simple array filling routine.

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 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 and 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$(X)
14 NEXT X
15 Z$=INKEY$: IF Z$="" 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$(X,Y)
15 NEXT Y
16 Z$=INKEY$: IF Z$="" THEN 16
17 NEXT X
```

Note: It is not necessary, on the Amstrad, 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. (By the way, the FILO rule applies to *all* functions which have a start/end format - FOR/NEXT, GOSUB/RETURN and WHILE/WEND.)

You will notice that the Z$=INKEY$ 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 6.9: 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$() array, while the inner loop controls the search of the rows of OB$() as we look for 'free' array space for the newcomers.

```
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 6.9. Routine to refill an array already in use.

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 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 (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)="O" 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 on 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 SAVEDGAME or whatever. After the game itself has been loaded, then, the player is asked whether he wants to play (1) 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.

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 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 the following 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 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 be that the writers 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 heaven 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 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 broad sword 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 happened when it struck 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 or 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 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 him that 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 pretty obvious example, and

not one that a writer is likely to overlook when he's 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 himelf 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, however, even though he tries to hide he will be told that 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 the other hiding places could either have been left open and the HIDE command would be obeyed (but useless). Alternatively all of the other barrels could have been 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 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 that they won't be noticed.

Program 6.10: 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 shortcuts, 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-defusible 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 'defusible 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!

```
1 REM *****  Random Lock  *****
2 :
3 :
8 REM ***  Set rnd value for lock
9 :
10 LO%=RND*900+100
20 LO$=STR$(LO%):LO$=RIGHT$(LO$,3)
30 MT=0
40 PRINT LO$
97 :
98 REM ***  Explain situation
99 :
100 Q=1:CLS:PRINT:PRINT "The safe has a
combination lock!"
107 :
108 REM ***  And allow up to 10 attempts
109 :
```

```
110 WHILE (MT-Q)<7 AND Q<4
120 PRINT:PRINT "Please enter digit numb
er";Q;" ";
130 TL$=INKEY$:IF TL$="" THEN 130
140 IF TL$<>MID$(LO$,Q,1) THEN PRINT:PRI
NT "Sorry - that was wrong." ELSE PRINT:
PRINT "Well done.":Q=Q+1
150 MT=MT+1:WEND
160 IF Q=4 THEN 200
167 :
168 REM *** Release booby-trap
169 :
170 PRINT:PRINT "Oops! After";(MT-Q+1);"
false tries a well placed booby-trap se
nds a block of stone down on your head.
  R.I.P.!!!":END
197 :
198 REM *** Or open safe
199 :
200 PRINT:PRINT "The safe is now open!!!
":END
```

Program 6.10. Random Lock routine (for would-be safe-crackers).

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 110 gets a random number and line 20 converts it into a string.

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

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

Line 110: 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 110–140: 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$. If they don't match, an error message is generated and the routine tries to REPEAT. If TL$ 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 150 and on to line 200. If, on the other hand, the player uses up to ten attempts without finding the right combination (notice MT building up in line 140) 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 alter lines 110 and 150 to read:

```
110 T=TIME+(SECONDS ALLOWED * 300): WHILE
    TIME<T AND MT<(MAXIMUM TRIES + 1) AND
    Q<4
.
.
.
150 WEND
```

And for a 'time only' controlled routine simply leave out the variable MT wherever it occurs.

Chapter Seven
We've Got Your Number

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 per cent 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. 5.6 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 that this was 'the only way' but rather that it was 'the only *efficient* way' of storing the required information. This takes us back to the 'calculated move' routine which I described in Chapter 5.

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 location – CR = CR +RL. And to go WEST or EAST you deduct 1 or add 1 respectively to your room number – 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 5.2, 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 and only 4 direction codes for each room (no UP or DOWN), this second method could use up 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 7.1: 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 around 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. Now the Amstrad allows you to store a zero in a DATA line without actually having a digit in the line as in:

```
1000 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 revalue all direction variables each time you enter a new room. 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 7.2 – 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 9.1 and 9.2 in Chapter 9) plus a full set of room descriptions and movement codes.

```
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 PL =
      N: GOTO 190
130   IF N$ = "S" AND S > 0 THEN PL =
      S: GOTO 190
140   IF N$ = "E" AND E > 0 THEN PL =
      E: GOTO 190
150   IF N$ = "W" AND W > 0 THEN PL =
      W: GOTO 190
160   PRINT : PRINT "I CAN'T DO THAT":
       GOTO X: REM *** X=START OF COMM
      AND INPUT ROUTINE
170  :
190   ON PL GOSUB 10000,10050,10100
200   GOTO X: REM *** X=START OF EVENT
       CHECK ROUTINE
```

```
9997 :
9998  REM *** START OF ROOM DESCRIPTI
     ONS
9999 :
10000  PRINT : PRINT "(ROOM DESCRIPTI
     ON)"
10010 N = 0:S = 11:E = 14:W = 0
10020  RETURN
```

Program 7.1. Direct revaluation of movement codes (numerical).

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: These lines 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 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 VAL(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 6.5 and 6.6). 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 7.2: A gentle squeeze

A second method, which involves *slightly* more complex programming, saves bytes by removing the need to re-set 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.

```
1 REM *****  Direct revalue #2  *****
2 :
3 :
90 REM ***  Routine to move to new
91 REM      location (when N$ is the
92 REM      section of CO$ containing
93 REM      a movement instruction)
95 :
96 REM [Note: DI cannot be used as a
97 REM        variable name in a
98 REM        working program]
99 :
100 IF LEN(N$)>1 THEN 2000
110 IF N$="I" OR N$="L" THEN 1800
120 IF N$="N" THEN DI=1:GOTO 170
130 IF N$="S" THEN DI=3:GOTO 170
140 IF N$="E" THEN DI=5:GOTO 170
150 IF N$="W" THEN DI=7:GOTO 170
160 PRINT:PRINT "I can't do that":GOTO X
:REM ***  X=start of command and input r
outine
170 NR=VAL(MID$(DI $,DI,2)):IF NR<1 THEN
 160
180 PL=NR
```

```
190 ON PL GOSUB 10000,10050,10100
200 GOTO X:REM ***  X=start of command a
nd input routine
9997 :
9998 REM ***  Start of room descriptions
9999 :
10000 PRINT:PRINT "(room description)"
10010 DI $="00111400"
10020 RETURN
```

Program 7.2. Direct revaluation of movement codes (using strings).

Line-by-line analysis

Although this second 'direct revaluation' program is longer than the routine in Program 7.1 it does save space in the long run because the actual movement codes – in DI$ – are shorter. Thus line 10010 of Program 7.1 takes up 22 bytes where line 10010 of Program 7.2 uses only 19 bytes (including the line header block in each case). As I said in the notes on Program 7.1, 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 7.2:

Lines 100–110: These lines are, as in Program 7.1, 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 7.1 – 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 has been collected execution moves on to line 170.

Line 160: Also as in Program 7.1 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 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 has 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 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 7.1.

Lines 10000 10020: These three lines are *basically* the same as the statements at the same line numbers in Program 7.1, *but* notice the difference in line 10010. Instead of assigning values to four variables we now have a single string called DI$. Thus we are still telling the computer that it cannot move North (N=0 in Program 7.1), that moving South takes the player to room 11 (E=11), and so on, but we do it by packing all the information into one variable value and let line 170 sort it out for us. Incidentally, if you were using this line in a game which included room numbers *above* 99 then this line would have to read 10010 DI$ = "000011014000"; that is, 000 for North, 011 for South, 014 for East and 000 for West.

You won't be surprised to hear that 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 (in a perfectly 'legal' manner, I hasten to add), so it will be of considerable value to start by discussing some of the details of how to use those PEEK and POKE operations I mentioned in the last chapter.

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. There is a way of controlling the Amstrad's memory *directly* rather than via BASIC and the computer's 'interpreter' (the main part of ROM – Read Only Memory – which turns BASIC instructions into a form that the actual processor can understand), but in order to do this we shall have to learn a bit (if you'll excuse the pun) about the Amstrad's use of the PEEK and POKE instructions.

What the hex going on?

If you're not much of a mathematician this next section may seem a bit confusing at first. But don't give up – hex can grow on you once you've grasped the basic idea and it is, after all, essential that you understand hex before you can use the commands we're going to be using later on.

So what is *hex* anyway? To explain it as simply as possible, think of the way that we usually count: 1, 2, 3, ... 9, 10, 11, ... 20, ... 30, and so on. This is usually referred to as the *decimal* system of numbers because it goes up in sets of 10. Thus if I write the number 5729 you know that I mean

$$\begin{array}{lll} & 5 \times 1000 & (= 5 \times 10 \times 10 \times 10) \\ \text{plus} & 7 \times 100 & (= 7 \times 10 \times 10) \\ \text{plus} & 2 \times 10 & (= 2 \times 10 \times 1) \\ \text{plus} & 9 \times 1 & (= 9 \times 10 \times 0.1) \end{array}$$

which equals five thousand, seven hundred and twenty-nine. Now, supposing that we – human beings – didn't normally have ten fingers and ten toes but *eight* fingers and *eight* toes. In that case we'd probably be counting in eights instead of tens! We wouldn't have the digits 8 and 9, because in this system of counting – known as *octal* and used by many of the earliest computers – the value 8 (decimal) would be represented by 10. And the *octal* value 5726 would mean

$$\begin{array}{lll} & 5 \times 1000 & (=5 \times 8 \times 8 \times 8) \\ \text{plus} & 7 \times 100 & (= 7 \times 8 \times 8) \\ \text{plus} & 2 \times 10 & (= 2 \times 8 \times 1) \\ \text{plus} & 6 \times 1 & (= 6 \times 8 \times 0.1) \end{array}$$

In other words, our octal number 5726 would represent the decimal

value 3032 (i.e. 2560 + 448 + 16 + 6). But notice that the numbers still *look* like decimal numbers.

Now, if the *base* value of any numbering system affects the point at which you add 1 to the left-most column (in decimal, whenever its right-hand neighbour totals 10; in octal, whenever its right-hand neighbour totals 8):

Decimal values	Octal values
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12

Fig. 7.1. Table of decimal and octal values.

What's going to happen if we use a numbering system with a value *higher* than 10? Obviously we're going to have problems, because we need to fit a value of ten, or more, into a *single* digit and yet the highest value digit we have is 9! I'm not just asking this question for fun – it really matters, because the *hex* (short for 'hexadecimal') numbering system works on base 16. In other words we need to be able to use a single digit to represent the values 10, 11, 12, 13, 14 and 15. The way that we do this is by using *letters* as well as numerals. Thus the first stage of the hex numbering system (with decimal equivalents) goes like this:

Decimal values	Hexadecimal values
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Decimal values	Hexadecimal values
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11

Fig. 7.2. Table of decimal and hexadecimal values.

And if we define the values of each *column* of a hex number (remembering that the highest value of any memory address in the Amstrad – or any other 8-bit computer – is 65535, &FFFF hex) then we get:

Column 4	*Column 3*	*Column 2*	*Column 1*
Value of digit × 4096 (16 × 16 × 16)	Value of digit × 256 (16 × 16)	Value of digit × 16	Value of digit × 1

Fig. 7.3. Table of hexadecimal digit values.

PEEKing and POKEing

The terms PEEK and POKE, when used in computing, actually mean exactly the same as in any other context though they refer specifically to something you are doing to a computer's memory. As you probably know, the Amstrad's memory is made up of 65535 'addresses', each of which contains a value between 0 and 255. When you define your own characters, for example (something we'll be looking at in detail in Chapter 11), you're actually storing *numbers* in addresses. Sometimes we may want to look at (PEEK at) the contents of a particular address or store a value at (POKE to) an address so that we can collect that value again later on. In this chapter I'll be showing you how to store your movement codes in a way that will save a good deal of space, and in Chapter 8 we'll be storing whole strings without the cost (in terms of RAM space) of creating an array. But let's start at the beginning.

In order to store or retrieve a *number* in memory we must start by assigning a value to a variable. Thus BA in Programs 6.5 and 6.6 *must* be initialised in the form:

```
20 BA=16384
```

Once you've done this you may both PEEK at the contents of address 16384 using

```
30 PRINT PEEK(BA)
```

or

```
30 NM=PEEK(BA)
40 PRINT NM
```

and alter the value in 16384 using either

```
30 PL=20
40 POKE BA,PL:REM NO brackets needed
```

or simply

```
30 POKE BA,20
```

And that's not all! Keeping BA as our base address we can PEEK at other addresses relative to 16384 using further variables or numbers. (If we couldn't then Programs 6.5 and 6.6 wouldn't be worth much!) To do this we simply add on the value of the relevant address, *minus* the value of BA, inside the brackets. Thus:

```
50 PRINT PEEK(BA+10)
```

will give us the value of the tenth location, or 'byte', after 16384 – that is 16394. So we can now establish a 'table' of movement codes in any 'legal' area of memory and then access any value we wish. The method of *retrieving* code values has already been illustrated in Programs 6.5 and 6.6. The only questions now are *where* are we going to store the codes, *how*, and how does this save on RAM space? After all, you've got to have the code values in your program in order to be able to POKE them into memory – haven't you?

The answer is yes. But if POKEing values into memory takes less space than when they are left as program lines or entered into an array, then obviously we don't want to waste the space we're saving by having the same set of values in *two* areas of memory. What *do* we do, then? This next program will reveal all.

Program 7.3: Now you see them - now you don't!

The one thing I haven't dealt with so far is how we decide on a suitable address for our base (i.e. the hex value in BA). In practice there is no one right place for BA – it will depend on the length of your program. The way we find BA, then, is as follows:

(1) When you enter *any* program which includes DATA lines then these lines should come at the *end* of your program. This makes 'debugging' (correcting your program) easier, and it also allows the program to RUN faster whenever it comes to a GOTO, GOSUB, etc. as it will search for the line it is going *to*, starting at the very first line of your program – if all DATA lines are at the end they won't be included in, and therefore won't slow down, that search process.

Of course, the movement codes won't necessarily be the only things you want to store as DATA – which brings us to the next question: do movement codes belong in the main program at all?

(2) In some computers we could store the movement codes as the *last* items of DATA in our program, restore them in compacted form, and then drop the end of program pointer so that that section of DATA was no longer recognised as part of our program. These pointers are not yet common knowledge in the Amstrad, but no matter, there are still at least two ways round the problem.

In order to store our 'packed' version of the movement codes in a safe place we have to know that the area we use will not later be used by the computer for some other purpose. We could make sure of this by storing them in the same place that they were taken from, overwriting the original DATA statements, but this wouldn't save any space at all so we can't relocate the pointer. It is much easier, and more effective, to store them at the top of RAM – just below the current location of HIMEM – and then relocate MEMORY just under the last code. The only problem now is that we have the entire set of codes taking up not one, but two, areas of memory, and we don't need the set that are stored as DATA. So why not make the 'Data Storer' into a separate program and then CHAIN the main program on afterwards? We'll still have our

codes stored safely above MEMORY (which is not affected by CHAIN), and look – no unwanted DATA! But we still don't now where BA will be.

(3) Fortunately, finding the location for BA is a very simple operation so long as it is handled correctly. We can find the top of usable memory by simply entering PRINT HIMEM as a direct command (i.e. without a line number). And the value we get will be the location for the *last* item in the movement code table. The *first* location for the table will, therefore, be at HIMEM minus the number of codes to be used. But that *isn't* BA! To find BA we now deduct from our second value the number of possible directions for movement, *plus* 1. Let's set this out as a practical example.

If the player could only move North, South, East and West, and the starting point for HIMEM were at 30000, and we were storing 400 movement codes (for 100 rooms), then our calculation would look like this:

Original position of HIMEM	= 30000	(decimal)
Area for movement codes	= 29600 – 30000	
Location for MEMORY	= 29599	
Position of BA	= 29595	(1st movement code address *minus* 1 *minus* number of directions for movement)

So now we know what to do, we can create a subroutine which will do the job for us.

```
1 REM  *****  DATA STORER  *****
2:
3:


10 TS=43903-TL: REM 'HIMEM' CAN'T BE
   USED LIKE A VARIABLE
20 FOR X=0 TO TL
30 READ MC
40 POKE TS+X,MC
50 NEXT X
60 TS=TS-1: MEMORY TS
70 B%=TS-4
80 CHAIN"(PROGRAM NAME)"
10000 DATA (MOVEMENT CODES)
```

Line-by-line analysis

Line 10: This first line sets up a value for TS – movement code Table Start – by deducting the number of movement codes (TL = Table Length) from the current value of HIMEM. Like several other values which are given in brackets, TL cannot be calculated by the computer but must be given a numerical value in the program itself.

Lines 20–50: A simple loop, controlled by the variable X, allows us to pass each movement code in turn to MC and then store it at the *relative* location indicated in line 50 – the value TS *plus* X. **Note:** Although the User's Manual *seems* to indicate that a memory location must be given as a hex value – &4000, for example – repeated experiments have shown that it works just as well with decimal values as used here.

Line 60: With our movement codes in place we now lower MEMORY in order to protect them from the computer which will use the area from MEMORY down for storage once the program proper is in action. Notice that we use the expression TS–1, not just TS, since the address given by MEMORY is itself a 'usable' location. Setting MEMORY to TS would allow the first unit of the movement codes to be wiped out.

Line 70: The value which we have previously referred to as BA must be stored as a numerical value. This is because no variables can be stored untouched when the CHAIN command is used. Of course, this means that all the other variables that we have used so far will now be entirely wiped out (but in this case we only erase MC, TS and X, which we have now finished with anyway).

Line 80: Lastly, we CHAIN the next program (or 'file' – see next chapter) and continue with the program.

Program 7.4: A moving experience

Finally, in the actual game program, we read off player's moves in exactly the same way that we handled the Random Item moves in Programs 6.5 and 6.6. Given that the directions for movement will need to be translated into a numerical value, we can use the following lines of code to handle any movement commands. As in Programs 7.1 and 7.2, this routine will not work on its own.)

```
1 REM *****  PLAYER MOVEMENT  *****
2 :
3 :
94 REM *** ROUTINE TO MOVE TO NEW
95 REM     LOCATION (WHEN N$ MAY BE THE
96 REM     SECTION OF A COMMAND INPUT
97 REM     CONTAINING A MOVEMENT
98 REM     INSTRUCTION)
99 :
100 IF N$ = "I" THEN 1900 ELSE IF N$ =
    "L" THEN 2000: REM HANDLE 'LOOK' AND
    'INVENTORY'
120 IF LEN(N$)>1 THEN N$=MID$(N$,4,1)
130 IF N$="N" THEN PM=1: GOTO 180
140 IF N$="S" THEN PM=2: GOTO 180
150 IF N$="E" THEN PM=3: GOTO 180
160 IF N$="W" THEN PM=4: GOTO 180
170 PRINT:PRINT"YOU CAN'T MOVE "N$: GOTO
    (START OF COMMAND INPUT ROUTINE)
180 PM=PL * PD + PM
190 NM=PEEK(BA+PM): IF NM=0 THEN 170
    ELSE PL=NM
200 ON PL GOSUB (PRINT A ROOM DESCRIPTION)
```

Line-by-line analysis

Line 100: The way that this routine is written assumes that the player has either entered a one-letter command or that his command began with GO followed by a blank space. We have also assumed that the only 'legal' one-letter commands are the directions – N(orth), S(outh), E(ast) and W(est), plus I(nventory) and L(ook). This line filters out the last two choices and redirects the program to the appropriate subroutines.

Line 120: This line deals with commands beginning with GO and extracts the fourth letter of the command which it expects to be the initial letter of a direction.

Lines 130–160: These handle the four 'legal' directions, assigning an appropriate numerical value to PM (Player's Move).

Line 170: If the program *hasn't* jumped to line 180 then the direction input is incorrect in some way, so an error message is displayed and the program goes back for a fresh command.

Lines 180–190: Taking PM as a base we now start to create a correct pointer for our search of the movement code table. PL (Player's Location) will already have been set elsewhere in the program and we

multiply this by the number of Possible Directions for movement (in this case 4) and add the value for the current *intended* move. We can now PEEK at a value in the movement code table (at BA + PM) and assign it to NM. If it is a value in the range 1–255 we 'move' the player to the new location and execution passes to line 200; otherwise, even though the direction was 'legal' in a general sense, it is obviously unusable at the player's current location so we go back for a fresh command via line 170.

Line 200: The Amstrad does *not* have the ability to handle *calculated* GOTOs or GOSUBs so we must use the ON...GOSUB instruction instead. By the way, if you ever run into a situation where you have lots of line numbers to fit into an ON...GOSUB instruction, why not break them down into 10s (or 20s) as in:

```
100 IF PL>10 AND PL<21 THEN 210
110 IF PL>20 AND PL<31 THEN 220
    .
    .
    .
200 ON PL GOSUB (10 LINE numbers):GOTO XXX
210 ON PL-10 GOSUB (next 10 LINE numbers):
    GOTO XXX
220 ON PL-20 GOSUB (next 10 LINE numbers):
    GOTO XXX
    .
    .
    .
```

Chapter Eight
A Code in Time Saves....

Leaving aside the 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 go 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 that the first adventure games were being developed – many of them on machines like the DEC PDP/10 – transistors had taken over from valves and had themselves, in turn, been overtaken by the silicon chip. Memory space in the average machine was increased 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 also made 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, then 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 computer with only 42.5K of program memory?'

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

(1) If you have a disk drive your problems are more or less over. Room descriptions may be stored in groups in sequential files. The size of each room description, and the total number of descriptions used in the course of a game, will then 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) Some computers will allow the room descriptions to be set up in a separate program which holds nothing but room descriptions (as DATA statements). These are transferred to an array and once the transfer is complete it is possible to SAVE the array alone. 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 compared with using a disk drive.

Time and space

Before we actually 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 then we can count on something in excess of 80K of storage space on each disk. This just happens to be, you will remember, exactly the size of the storage space we estimated would be 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. 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 – all on the same disk. In this way the computer can 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 only need enough space in RAM to hold one room description and one set of movement codes, leaving almost all of 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 directory sectors and on 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

really worth any 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 what would seem to be a major limitation since it requires that *all* the room descriptions be in memory at all times. So we come back to our original question: how do we get all that text into a limited amount of RAM space?

Well, 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 really is 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 just like to give you some idea of 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 a little over 9000 characters. So let's suppose that we can only encode, or tokenise, just 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*, *then*, *rather*, *other*, etc., etc. – so it's not much of a surprise to find that it occurs about 146 times (including a few The's) in this passage. In other words, out of 9000 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 passage 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 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 that 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 that 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, but 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 8.1: Sardines - the computer version

So how do we go about this 'text crunching'? First of all we need to study the ASCII codes for the Amstrad. 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. These codes all come *below* 127, so we can use values above 127 for our codes without running the risk of fouling up the computer's normal operations. What we are going to do consists of 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.)

```
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 8.1. Demonstration of the encoding/decoding process.

Line-by-line analysis

Line 10: In the two later programs this string would be the one that 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 8.2) and subsequently stored as CD$(X).

Lines 30–70: A simple search loop which 'steps' through the string looking for a match to A$(1). It moves one character at a time, but always reads two characters at a time (X and X+1). If a match is found then a code character is added to B$ and the search jumps forward *two* letters – one letter in line 30 (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 just 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 8.2: 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 because the 'drag factor' is the constant searching of the SA$() array for matching letter groups.

```
1 REM *****  String 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 CLS:PRINT:PRINT "Please enter string
 number";Y;"now: ":PRINT
130 IN$=""
140 FOR X=1 TO 255
150 I$=INKEY$:IF I$="" THEN 150
160 IF I$=CHR$(13) THEN IN$=LEFT$(IN$,X-
1):X=255:GOTO 190
170 IF I$=CHR$(127) THEN PRINT CHR$(8)+C
HR$(16):IN$=LEFT$(IN$,X-1):X=X-1:GOTO 19
0
180 IN$=IN$+I$:PRINT I$;
190 NEXT X
196 :
197 REM ***  Display IN$ for
198 REM        corrections
199 :
200 CLS:PRINT:PRINT IN$:PRINT
210 PRINT:PRINT "Is this correct (Y or N
)? ";
220 Z$=INKEY$:IF Z$="" THEN 220
230 PRINT Z$:IF Z$<>"Y" THEN 120
240 PRINT:PRINT "Thank you."
297 :
298 REM ***  And break it down
299 :
300 MF=0: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)=SI(HA)+1:
HA=CA:MF=1
340 NEXT HA
350 IF MF=0 THEN SA$(CA)=HO$:CA=CA+1 ELS
E MF=0
360 NEXT Q
397 :
398 REM ***  Get another string?
399 :
400 PRINT:PRINT "Any more strings to ana
lyse (Y or N)? ";
410 Z$=INKEY$:IF Z$="" THEN 410
420 IF Z$<>"Y" THEN Y=255
430 NEXT Y
497 :
498 REM ***  Or display results
```

```
499 :
500 FOR Y=1 TO CA STEP 15
510 CLS
520 FOR X=Y TO Y+14
530 PRINT SA$(X),SI(X)
540 NEXT
550 PRINT:PRINT "Press any key to contin
ue ";
560 Z$=INKEY$:IF Z$="" THEN 560
570 PRINT " ":NEXT Y
```

Program 8.2. String analysis routine.

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. The variable CA is used in lines 300–370.

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 this 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 letter of the string, setting the value of X to X−1 and then re-entering the loop.

Lines 200–240: These lines present the completed (?) string for approval. Line 240 is there just to reassure you that the program is still in operation as 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 will crash. 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$. If a match is found then the counter for that element – SI(HA) – is incremented, HA is set to its highest possible value (i.e. CA) and the program jumps to complete the loop at 360 so that the Q loop can go again. If no match is found then the first empty element of SA$() is filled, CA is incremented and the Q loop is repeated if necessary.

Lines 400–430: These ask if there are any more strings to be analysed. If the answer is 'Yes' then the Y loop goes again. If not, 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 8.3: CRRUNCCCH!

Having analysed the contents of your room descriptions you might well choose to 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 also have to check whether 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 just chopping off lines 300 onwards to prepare for this next program.

```
1 REM *****  String encoder  *****
2 :
3 :
```

```
8 REM ***  Prepare storage arrays
9 :
10 DIM RD$(255),CD$(127):LC=127
17 :
18 REM ***  And fill code array
19 :
20 FOR X=1 TO LC
30 READ CD$(X)
40 NEXT X
97 :
98 REM ***  Input strings
99 :
100 FOR Y=1 TO 255
120 CLS:PRINT:PRINT "Please enter string
 number";Y;"now: ":PRINT
130 IN$=""
140 FOR X=1 TO 255
150 I$=INKEY$:IF I$="" THEN 150
160 IF I$=CHR$(13) THEN IN$=LEFT$(IN$,X-
1):X=255:GOTO 190
170 IF I$=CHR$(127) THEN PRINT CHR$(8)+C
HR$(16):IN$=LEFT$(IN$,X-1):X=X-1:GOTO 19
0
180 IN$=IN$+I$:PRINT I$;
190 NEXT X
196 :
197 REM ***  Display IN$ for
198 REM         corrections
199 :
200 CLS:PRINT:PRINT IN$:PRINT
210 PRINT:PRINT "Is this correct (Y or N
)? ";
220 Z$=INKEY$:IF Z$="" THEN 220
230 PRINT Z$:IF Z$<>"Y" THEN 120
240 PRINT:PRINT "Thank you."
297 :
298 REM ***  And insert codes
299 :
300 FOR X=1 TO LC
310 WHILE Y<>0
320 Y=INSTR(IN$,CD$(X))
330 IF Y>0 THEN LE=Y-1+LEN(CD$(X)):IN$=L
EFT$(IN$,Y-1)+CHR$(X+127)+RIGHT$(IN$,LEN
(IN$)-LE)
```

```
340 WEND
350 NEXT
396 :
397 REM ***  Store encoded string
398 REM      and go round again?
399 :
400 RD$(Y)=IN$
410 PRINT:PRINT "Encode another string o
r quit (enter A  or Q)? ";
420 Z$=INKEY$:IF Z$="" THEN 420
430 IF Z$="Q" THEN END
440 IF Z$<>"A" THEN 410
450 PRINT " ":NEXT Y
1000 DATA (codeable letter groups)
```

Program 8.3. String encoder routine.

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. 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 possible value is 127 and should be set in line 10.

Lines 100–240: Exactly as in the string analyser program.

Lines 300–350: The outer loop in this section (FOR X = , etc.) takes us through the entire list of encodable letter groups in the CD$( ) array. Thanks to the existence of the INSTR( ) function in LOCO' BASIC we can encode letter groups where they are all of the same length *or* of mixed lengths – all at the same time!

Lines 310 and 340: Because an encodable letter group may appear more than once in any string we use a WHILE... WEND loop to keep searching through the string until Y = 0 (i.e. until no match for CD$(X) can be found).

Line 330 does the actual encoding when a match for any part of CD$( ) is found in IN$. But notice that we *add* 127 to the value of X to get a correct code value.

Line 400: Stores the encoded string as RD$(Y).

Lines 410–450: You now have two options: to continue encoding – by entering a new string – or to QUIT. Line 430 handles QUIT, line 440 flushes out any 'wrong' choices, and line 450 re-activates the Y loop to go back for a new string.

'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 *simple*, but then it isn't all that difficult, either. So before I give you the answer, and bearing in mind the kind of operation we used in the last chapter, perhaps you'd like to have a go at writing the routine yourself. Here are the main points that the necessary code must deal with:

(1) Reset MEMORY to allow space for the stored strings in a 'safe' area.

(2) Working your way *backwards* through memory, store the *last* string at the top of memory (in other words work your way down from 43903).

(3) Once the string is stored take a note of the address of the first byte of memory for that particular string and save it in Hi-byte, Lo-byte form. (If you don't understand what I mean by 'Hi-byte/ Lo-byte' don't worry – all will be explained in a moment.)

(4) If you've any string left to store then go back to step (1).

(5) When all the strings have been stored enter a 'table' of byte pointers below them so that, working your way *up* in memory, you have a list of the addresses for the start of each string in Hi-byte–Lo-byte form.

(6) Make a note of the address of the *start* of this table – its length will be (number of rooms)*2 – then SAVE that whole area of memory in the normal way as though you were saving a file (see Chapter 2, page 7 of the User's Manual).

To use this method your *game* program must contain the following items.

(1) As soon as the program starts RUNning it *must* set MEMORY to the byte below the array table, or everything at the top of the programming area will be overwritten. (**Note:** When resetting MEMORY it's quite a good idea to follow the operation with the CLEAR instruction to ensure that the computer recognises the new MEMORY location.)

(2) In order to reclaim any string, you will need a printout loop which is set to read LEN(RD$) letters where RD$ is found at the address given by high byte * 256 + low byte. This looks pretty complicated at first sight so let's take a concrete example. Here I've assumed that the 'pointer table' starts at 15000 (decimal), and we want to find Room Description number 20 (the variable PL – Player's Location – should be set to 20 at this point as this is the room he has just moved in to):

```
10 K=14998 + (PL * 2):IN$="":PT=0
20 RD=PEEK(K) * 256:RD=RD + PEEK(K+1)
30 WHILE IN$<>(code character)
40 IN$=CHR$(PEEK(RD+PT)):IF IN$<>(code
   character) THEN RD$=RD$ + IN$
50 PT=PT+1:WEND
```

And there you are – it really isn't so difficult after all.

'Not so fast!', says the voice. 'What's this code character stuff?'

So far we've been using codes as substitutes for letter groups; now we want a single code which simply signals the end of a coded string. The point is that only the BBC Micro and Electron have the capacity to store a string directly into memory given only the address of the first byte. Since this is such a useful function, especially when preparing adventure games, I've created a simple BASIC imitation. All we need, therefore, is a starting address and we can read off any string without needing to know its length (which all helps to save time when the program is RUNning).

Given the purpose of this particular code character the obvious choice is CHR$(13) – which the computer interprets as <RETURN>, the command to finish printing on one line (of the screen or printer) and to move to the next line. We could, of course, have used the question mark, interpreted by the computer as a PRINT command in the right context. The trouble is that a display printout may legally contain a ? symbol as part of the display. So let's stick with CHR$(13).

In this case all encoded strings must end with CHR$(13) – *before* being stored. To do this, then, line 400 of our program should now read:

```
400 RD$ = IN$ + CHR$(13)
```

Next we must set aside enough RAM space to hold all the encoded strings plus the 'look-up' table. Obviously, we won't know in advance how much space we're going to need so I've chosen to leave 20K for

the program area and to use the remaining 22.5K as storage. To do this we add a new line to the start of the program:

```
5 MEMORY = 20863: CLEAR: TS=43906
```

By the way, note that MEMORY is not a variable in the normal sense so it will not be corrupted by the CLEAR command.

Since we are no longer planning to store all of the room descriptions in an array we no longer need the RD$() array, but we will need a temporary array for our pointers and a control variable for that array. Line 10 should now read:

```
10 DIM TS(500), CD$(127): LC=127: T=-2
```

Next, line 400 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 400 should now look like this:

```
400 GOSUB 2000
```

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 T=T+2: SP=1
2010 FOR X=(TS-LEN(IN$)) TO TS
2020 POKE X,ASC(MID$(RD$,SP,1):SP=SP+1:
     NEXT
2030 ST(T)=TS MOD(256):ST(T+1)=INT(TS/
     256):TS=TS-(LEN(RD$)+1)
2040 PRINT:PRINT "TS ="TS
2050 RETURN
```

In line 2000 we always increment the value of T (our storage-address-pointer array variable) by 2 so that it will start at 0 and move up to 2, 4, 6, 8, etc., when used in line 2030. SP gives us the character in RD$ to be POKEd next.

Lines 2010–2020: Here we use a straightforward loop to POKE RD$ into memory, one character at a time. Note that the string is POKEd in the order that it would print out, being read off using SP. On the other hand, the complete strings are POKEd in in reverse order. That is to say, if you're storing 100 strings then store string 100 first, then 99, then 98 and so on.

Line 2030: Now we need to save a record of where abouts in memory our string is, and we do this, for the moment, by using the ST() (Storage Table) array. But we do it in a slightly unorthodox manner in

that it starts out its life by reading *backwards* – you'll see why when we come to the next subroutine. For the moment all we need to know about is the Hy-byte/Lo-byte system I mentioned earlier.

If you think back to my comments on hexadecimal numbers in the last chapter you will remember that we said the highest value for an address in the Amstrad was 65535. This is because 65535 is the highest number that can be stored in *two* bytes. I'll go into this in a bit more detail in Chapter 11, but in the meantime suffice it to say that you cannot store a number bigger than 255 in a single address (i.e. a single byte). But the *lowest* point in our room description area is 20864, and it goes all the way up to 43903! In order to store numbers like these we have to break them down into two smaller numbers, and this is what line 2030 does. IN ST(T) we enter the *integer* (whole number) value of the *remainder* when TS is divided by 256 (this is known as the 'Lo-byte' of the total value), and in ST(T+1) we enter the integer value of TS divided by 256 (i.e. the 'Hi-byte' since it represents the higher or larger part of the total value of the two bytes when they are combined). In other words, the true value of TS is ST(T+1) multiplied by 256 + ST(T). In concrete terms, if we were storing the number 14576 then ST(T+1) would contain the value 56, while ST(T+1) would contain the value 240. Try entering the following on your computer and you'll see what I mean.

```
PRINT 56 * 256 + 240
```

The last thing we do in line 2030 is to deduct 1 *plus* LEN(RD$) from the current value of TS. This ensures that, when we come to store the next string, it ends at the byte (address) immediately *below* the string just stored and does not overlap its first letter.

Line 2040: Obviously we cannot know in advance exactly how much storage space we are actually going to need for our room descriptions, and it's possible that we might outgrow the 22.5K allowed for here. In order to avoid possible disasters caused by overwriting the working space between HIMEM and MEMORY I've included this line so that you know where TS is after each string has been stored. To be on the safe side, don't enter another string if the length of the next string plus 20864 plus the number of strings you've entered ×2 is greater than the value of TS. I realise that this could be very frustrating, especially if you've almost completed your task, but it's better to save what you've got than to lose everything. Once you've got the hang of these routines you shouldn't have too much trouble in saving one session's work, altering a few of the values in the programs, and then RUNning them

again so as to complete the task. On the other hand you may find that you *have* to start again from scratch because continuing on the same course won't leave enough room for the movement codes and the game program.

Line 2050: Lastly we RETURN to the main program. You can now choose whether to enter another string or quit.

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. If we are ending the encoding routine 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 3000: END
```

and then we add the 'Table Maker' subroutine as follows:

```
3000 FOR Q=0 TO T
3010 TS=TS-Q
3020 POKE TS,ST(Q)
3030 PRINT TS: IF TS=(VALUE OF MEMORY -
     1) THEN PRINT: PRINT"OUT OF STORAGE
     SPACE": END
3040 NEXT
3050 PRINT: PRINT"STORAGE TABLE COMPLETE
     .": PRINT: PRINT"START OF TABLE IS
     AT"TS
3060 RETURN
```

If you've taken the necessary precautions the Table Maker routine should run correctly, leaving a complete set of room descriptions and the pointer table safely protected at the top of memory.

The only item of any interest in this routine is the way that we work backwards and forwards at the same time in lines 3000–3010. The 'Q loop' obviously increments by 1 each time, and by deducting this value from TS – which was left pointing at the address *below* the first string at the end of the last routine – we manage to read *up* through the ST() array, but store the values *downwards* in memory. And now we have just one more job to do – SAVE all our hard work for future use.

If you read Chapter 2 of the User's Manual carefully you will see that you don't have to include an *execute* address in a binary SAVE, so all that is needed to SAVE the block of memory is:

```
SAVE"(FILE NAME)",B,SSSS,FFFF
```

where SSSS is the hex value TS when the encoding program ended, and FFFF is the standard value of HIMEM - MEMORY. One last point before we finish this section: this same 'save memory' process could, of course, be used to SAVE the movement code table.

Programs 8.4 and 8.5: Bringing it all back home

And that's just about it. Our last two programs, positive midgets compared to the string analyser and the encoder, are different versions of the kind of decoding routine to be included in the actual adventure game. Program 8.4 deals with encoded text held in an array and Program 8.5 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 8.4 won't run quite as fast as the demo program (its actual speed will depend on the size of the RD$() array). Nor, on the other hand, will they run anywhere near as slowly as the analyser and encoder programs, since we are able to look quite directly at both the contents of RD$(PL) and CD$() rather than searching through the arrays as we have a specific reference to the location of the relevant information in both.

Program 8.4

```
1  REM *****  DECODER #1  *****
2 :
3 :
4  REM  *** RD$() - 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 (RD$(PL))
110 K =  ASC ( MID$ (RD$(PL),Y,1)
```

```
120  IF K > 127 THEN  PRINT CD$(K - 1
     27);: GOTO 140
130  PRINT  MID$ (RD$(PL),Y,1);
140  NEXT
150  RETURN
```

Program 8.4. Routine to decode a string held in an array.

Line-by-line analysis

Line 100: It is assumed that the room descriptions will finally be stored as elements of RD$(). RD$(PL), where PL stands for the player's current location, 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 RD$(PL) – unless, of course, the value passed to K is greater than 127. In that case we deduct 127 from the value of K and use the result as an index to the RD$() 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 RD$(PL) has been printed). We then close off the string with a blank space (since it was left open in line 130) and RETURN from what should indeed be a subroutine since it will be used so frequently.

Program 8.5

Program 8.5 is the routine we need to decode room descriptions which have been previously POKEd into memory – *with* a look-up table. I've actually included an alternative version of the main section of this program which should enable it to run just a little faster as the calculation element is slightly briefer. However, more of that after we've seen the program itself.

```
1 REM *****  DECODER #2  *****
2 :
3 :
4 REM *** BA=Table start-2
5 REM     PL=Player's location
6 REM     CD$()=Code letter groups
```

```
7 REM      This subroutine is used with
8 REM      each successful movement
9 :
17 REM *** Initialise BA at
18 REM       start of game
19 :
20 BA=(first table byte)-2
97 :
98 REM *** Read 2 bytes from table
99 :
100 TL=PL*3+BA
110 RD=PEEK(TL)*256+PEEK(TL+1)
117 :
118 REM *** Then output string
119 :
120 WHILE K<>13:REM *** 'EOL' code
130 K=PEEK(RD)
140 IF K>127 THEN PRINT CD$(K-127);:GOTO
    160
150 PRINT CHR$(K);
160 RD=RD+1:WEND
170 RETURN
```

Program 8.5. Routine to decode a string stored in memory.

Line-by-line analysis

Line 20: If you are using any variables rather than entering a number (the Amstrad can actually 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 2 (if PL=1 then PL * 3 = 3). We must therefore 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) $\times$ 2 + 10495 BA to get the real start of the table.

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 *two* bytes in the table we can now get

the values needed to calculate the starting address of the string that we want, and collect the string itself.

Lines 120–160: From this point on the program is very different to Program 8.4. The difference – and what a difference! – is that we have saved a good deal of space that would have been used (by the computer's operating system) if we had used an array to hold our room descriptions. Notice that both print statements end with a *semicolon* to ensure that we print *across* the screen, not down.

In relation to this method of handling the strings I should point out that we could use a rather different method of handling the routine in these lines, as follows:

```
120 FOR X=0 TO 255
130 K=PEEK(TL+X):IF K>127 THEN PRINT
    CD$(K-127);:GOTO 160
140 IF K=13 THEN X=255:GOTO 160
150 PRINT CHR$(K);
160 NEXT
```

Here we PEEK at each byte of the string in turn, taking the ASCII value directly from memory rather than having to convert it – compare this version of line 130 with that in the original program. Obviously this saves a certain amount of time, but unfortunately we lose some of it again by having to check for the RETURN character (signalling the end of the string) *every* time we go round the loop.

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 7.1 notes) or to the command input routine, according to the method you have selected.

And talking of command input routines ...

Chapter Nine

The Well-chosen Word

The Inglish language (that *is* spelt correctly) was especially 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 you will probably already know that the entire English language, in its spoken form, can be broken down into just 64 *phonemes* or sounds. The great complexity of actual speech is dependent on 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 in which 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 another three. 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 word 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 elocution, I admit, but it is possible to store simple messages like 'Hello. I am an Apple computer' plus the record/playback program, in about 3.25K!

If only parsing written commands were that easy. But it isn't. For in order to have the computer 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 equation would make about as much sense as the old favourite: 'How long is a piece of string?' Since we are talking about computing, however, it makes very good sense indeed.

The biggest problem that we face when we try to get a computer to understand plain English is in the area of *implied* meaning, as I suggested just now. If we convert our bricklayer sentence into a similar kind of BASIC statement you'll see what I mean:

```
10 IF X = 8 OR 12 GOTO 200
```

At first glance this statement looks perfectly all right. Unfortunately it isn't, and it contains two of the most common errors made by newcomers to program writing: (a) the assumption that once you've mentioned which variables you want to deal with the computer will realise that future conditions or instructions also apply to the same variable, and (b) that it should execute the GOTO instruction *after* finding a favourable value for X. In fact, as someone once said, a computer is really nothing more than a high speed idiot. It can only do *exactly* what you tell it to do – no more and no less. It assumes *nothing*, not even the fact that you want it to act according to two possible values of a given variable. What we should have written, of course, was:

```
10 IF X = 8 OR X = 12 THEN GOTO 200
```

Ironically, however the word GOTO could be omitted as this is one implied command that the computer *can* understand.

So if the computer doesn't understand its own language, so to speak, how much more difficulty will it have in understanding a complicated human language?

The original adventure programs, despite being run on mini-computers with around 256K or more of RAM, could still only accept 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.

The long and the short of it

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 will find it tiresome to have to enter every command *in full* – especially when they are rerunning 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 9.1: 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 but 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 Word 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 redefinition 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? The process is actually pretty simple, using a short formula given below. As you can see in this program, the operation breaks down into just four stages:

(1) The program first compares each letter with a list of valid command letters in order to try to get a positive value for VP (Verb Place) and NP (Noun Place).

(2) In this simplified example, if either of the characters in CO$ is not an upper case letter or if a match cannot be found in VE$/NO$ 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 values of VP (for the first letter in CO$) and NP (for the second letter). From these two values, plus EN (a 'constant' which equals the total number of nouns *plus* one, i.e. EN=17), we can find a unique value for every command by using the formula:

```
X = VP * EN + NP - EN
```

Note: The second EN is used to make sure that the possible values of X start from 1 and not from EN+1. (If VP=1 and NP=1 and EN=17 then X=VP*EN+NP would produce the result X=18.) In our formula EN is deducted again so where VP=1 and NP=1 and EN=17 the result of X=VP*EN+NP–EN will be X=1. (By the way, the Amstrad will always try to perform a multiplication before it performs an addition or subtraction, whilst plus and minus ops have equal priority and will be dealt with *in order, from left to right*, unless brackets are used to alter the 'order of precedence' – calculations inside brackets are always executed first. Thus our formula will be executed in exactly the same order as the sections appear – VP will be multiplied by EN, the value of NP will be added to the first result and then EN will be deducted to give the final value to be assigned to X.)

(4) Having found a value for X we can, finally, use this to calculate a second value as in the following statement, for example, to move to the correct line to execute every possible command:

```
ON X GOTO (or GOSUB) (range of line numbers)
```

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 combinations will be directed to line 3000 (for example), which simply prints out 'I CAN'T DO THAT' and then goes back for a new command.

```
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$="ABCDEFGHIJ":NO$="ABCDEFGHIJKLMNO
P"
90 BA$=CHR$(8)+CHR$(16):REM ***  Backspa
ce/delete characters
96 :
97 REM ***  Convert CO$ to two
98 REM       numerical values
99 :
100 PRINT:PRINT "What now? ";
110 V$=INKEY$:IF V$="" THEN 110
120 VP=INSTR(VE$,V$)
130 IF VP=0 THEN PRINT" ":GOTO 300
140 PRINT V$(VP);
156 :
157 REM ***  Handle valid 1 letter
158 REM             commands
159 :
160 IF VP=2 THEN PRINT " ":GOTO 3000:REM
 ***  Go to 'look' routine
170 IF VP=9 THEN PRINT " ":GOTO 3500:REM
 ***  Go to 'inventory' routine
180 N$=INKEY$:IF N$="" THEN 180
190 IF N$=CHR$(127) THEN FOR X=1 TO LEN(
V$(VP)):PRINT BA$;:NEXT:GOTO 110
200 NP=INSTR(NO$,N$)
210 IF NP=0 THEN PRINT" ":GOTO 310
220 PRINT " ";N$(NP);
```

```
247 :
248 REM *** Allow correction
249 :
250 AL$=INKEY$:IF AL$="" THEN 250
260 IF AL$=CHR$(127) THEN FOR X=1 TO LEN
(N$(NP)):PRINT BA$;:NEXT:GOTO 180
270 PRINT" ":GOTO 400
297 :
298 REM *** Deal with illegal CO$
299 :
300 PRINT:PRINT "I don't understand ";V$
:GOTO 100
310 PRINT:PRINT "I don't understand ";V$
(VP);" ";N$:GOTO 100
397 :
398 REM *** Process valid CO$
399 :
400 X=VP*EN+NP-EN
410 ON X GOTO A,B,C:REM *** Where comma
nd handling routines start at A,B,C,etc.
9997 :
9998 REM *** Command data [demo only]
9999 :
10000 DATA GET,LOOK,CLIMB,DROP,EXAMINE,T
HROW,GO,SAY,INV,WEAR
10010 DATA THE TREE,THE GLOVES,THE BANAN
A,THE RING,THE SPADE,HELLO,YES,THE SWORD
10020 DATA THE BOOK,HEEBEE JEEBEE,NORTH,
SOUTH,EAST,WEST,UP,DOWN
```

Program 9.1. Two-letter command interpreter.

Line-by-line analysis

Line 10: This creates the two arrays which will hold all the verbs (in V$()) and nouns (in N$()) used by this routine in their full form. EN represents the number of nouns (16) plus 1 for the calculation in line 400.

Lines 20–70: Two simple loops to fill V$() and N$(). Of course, if the

two lists had been the same length a single loop would have been sufficient, but the items in the DATA statements (lines 10000–10020) would have to be rearranged accordingly.

Lines 80–90: Three strings are set up – the verb string, the noun string and the 'delete' string. The first two each contain a list of all the *valid* letters for verbs and nouns respectively.

Lines 100–140: Having asked for a command input we now GET the first letter. This is then converted to a numerical value, held by VP, which will be a positive number if a match for V$ is found in VE$, or 0 if no match is found – in which case we display the error message in line 300. If we obtain a successful match the appropriate verb in V$() is displayed (*not* V$). Notice that the Amstrad recognises V$ by itself and V$(), an 'arrayed' variable, as quite separate entities.

Lines 160–170: This handles V$="I" and V$="L" which are recognised, through the value of VP, as being the two 'one-letter' commands.

Lines 180–220: The second input letter, N$, is dealt with in much the same way as V$ was handled – except that here (line 190) if N$ is the DELETE key (i.e. if N$=BA$) then the *verb* input is deleted and the command input routine starts all over again.

Incidentally, although we are actually GETting each letter individually it will appear to the user that he is being allowed to enter a single string (as in an INPUT statement).

Lines 250–270: The player is allowed to delete his chosen noun using the same process as in line 190. If AL$ is *anything* except the DELETE key the program goes on to line 400 to try to execute the command.

Lines 300–310: Print out an error message – including a display of the incorrect input.

Lines 400–410: Perform the 'calculated GOTO' which I have already discussed.

Lines 10000–10020: Finally, we have the verbs and nouns that go into V$() and N$() and are displayed on screen. Thus we have a complete, though simple, command input routine which is 'ready to RUN' (though it won't actually *execute* any commands until you add on the appropriate routines, of course).

Wasn't that fun? Well, it wasn't *that* bad! But I did start off by talking about the kind of advance 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 that 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 through 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 is not only written in full but also 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 of these 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 the use of 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 that 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, a one-ton jar of marmalade or anything else you can lay your hands on.

Using our new super compound parser you might think that we would 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. For 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. (This last process is *not* included in the program but you will find a full explanation in the line notes which follow.)

Program 9.2: 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 whole lot clearer when you see how they are dealt with in practice.

```
1 REM *****  Compound parser  *****
2 :
3 :
8 REM ***  Set up word array
9 :
10 DIM CO$(26),A$(30):VT=10:NT=26:EN=17
20 FOR X=1 TO 26
30 READ CO$(X)
40 NEXT:BA$=CHR$(8)+CHR$(16)
496 :
497 REM ***  Get CO$ as a set of
498 REM        words and symbols
499 :
500 X=1:A$(X)="":PRINT:PRINT "What now?
";
510 IF X>1 AND (A$(X-1)="" OR A$(X-1)="
") THEN X=X-1
520 A$=INKEY$:IF A$="" THEN 520
530 IF A$=CHR$(127) THEN PRINT BA$;:A$(X
)=LEFT$(A$(X),LEN(A$(X))-1):GOTO 520
540 IF A$=CHR$(13) THEN PRINT " ":CH=1:G
OTO 600
550 IF (A$=" " AND X>1) AND (A$(X-1)=","
 OR A$(X-1)=".") AND A$(X)=" " THEN PRIN
T A$;:GOTO 510
560 IF A$=" " THEN PRINT A$;:X=X+1:A$(X)
="":GOTO 510
570 IF A$="," OR A$="." THEN PRINT A$;:G
OTO 510
580 PRINT A$;:A$(X)=A$(X)+A$:GOTO 520
597 :
598 REM ***  Verb search
599 :
600 W=CH:WF=0:FL=0
610 FOR VS=1 TO VT
620 IF A$(W)=CO$(VS) THEN VP=VS:VS=VT:W=
W+1:FL=1
630 NEXT VS:IF FL=1 THEN FL=0:GOTO 700
640 IF W<X THEN W=W+1:GOTO 610
650 PRINT:PRINT "Commands must start wit
h a valid verb.":GOTO 500
```

```
697 :
698 REM ***  Noun search
699 :
700 FOR NS=VT+1 TO NT
710 IF A$(W)=CO$(NS) THEN NP=NS:NS=NT:FL
=1:GOTO 730
720 IF A$(W)=RIGHT$(CO$(NS),LEN(A$(W)))
THEN B$=A$(W-1)+" "+A$(W):NS=NT:FL=2
730 NEXT:IF FL=1 THEN FL=0:GOTO 2040 ELS
E IF FL=2 THEN FL=0:GOTO 2000
740 IF W<X AND WF=0 THEN W=W+1:GOTO 700
750 IF WF THEN 600
760 PRINT:PRINT "If you've included a va
lid noun I sure  can't find it!!":GOTO 5
00
797 :
798 REM ***  Process command
799 :
800 REM EV=VP*EN+NP-EN
810 ON EV GOTO A,B,C,etc.
1997 :
1998 REM ***  Check adjective
1999 :
2000 FOR QS=VT+1 TO NT
2010 IF B$=CO$(QS) THEN NP=QS:QS=NT:FL=1
2020 NEXT:IF FL=1 THEN FL=0:GOTO 2040
2030 PRINT:PRINT "Sorry - there is no ";
B$:GOTO 500
2040 PRINT:PRINT "O.K.":GOTO 800
2050 IF W<X THEN 20000
9997 :
9998 REM ***  Command data [demo only]
9999 :
10000 DATA GET,GO,TAKE,READ,DROP,OPEN,UN
LOCK,ENTER,THROW,KILL
10010 DATA EAST,E,WEST,W,NORTH,N,SOUTH,S
,DOOR,WINDOW
10020 DATA GREEN BOOK,KEY,ROCK,KNIFE,GOB
LIN,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 9.2. Routine to handle compound command input.

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 exercises what I think is a reasonable limit on the amount of instructions which 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 the 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 as 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: This is the second entry point into the routine and is 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: This deals with deletions made during input. The deletion is made both in screen display and in the element of A$() currently being entered.

Line 540: This line comes into play when the ENTER 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: Any blank spaces entered for A$ after a comma or a full stop are handled. 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: This handles commas and full stops *only* as valid input, assigning them to their own elements of A$(). Any other punctuation marks would, unless deleted at the time of entry, be included as part of the preceding word and would certainly invalidate the entire command string from that point on if attached to a verb or a noun. Finally line 570 handles any other input, either letters, numbers or symbols, adding them to the current element of A$(). 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 610–640: 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 650: ... prints a carefully worded rejection and refuses *all* of the current command input. This may seem a bit drastic, but it does save the player from finding that, for example, because he wasn't able to GET THE SWORD he now has to try to KILL THE GOBLIN with his bare hands.

Lines 700–730: A simple loop which searches through A$() for a noun. In this case I've allowed the search to include a match with the last LEN(A$(W)) letters of any of the nouns to cope with elements which include an adjective for clarity (see GREEN BOOK in line 10020). So, if A$(W) matches one of the elements in CO$() *exactly* we can move straight on to line 2030. If we only have a *partial* match then a new string (B$) is created by adding the previous word (A$(W–1)) *plus* a blank space to the beginning of A$(W) and the program goes off to line 2000 for a second noun search.

Line 740: Ensures that this search continues until either we find a noun, the end of A$() is reached, or ...

Line 750: ... if 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 760: 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. Notice, however, that we do *not* alter the value of X here, as it may be needed later if the input string contains more than one command. In fact, if X is being used in this routine it shouldn't be used anywhere else in your program except if defined as a LOCAL variable inside a PROCedure.

Lines 2000–2030: This subroutine is designed to find a 'noun' in CO$() which includes an adjective – remember my example of 'THE LEFT

DOOR"? When we left the noun search routine to come to this routine we created a new noun string, B$, as what we are looking for is a value for NP. Losing the current A$(W) doesn't really matter, therefore. These lines simply repeat lines 700, 710 and 730 using the same material but with a couple of different variable names. 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 the command can be executed and the program should be sent to line 800. Line 2050 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 command input (in other words: does W=X yet)?

To see this routine in action using input with more than one command change line 800 to GOTO 2050 and then RUN the program. If you actually want to see what the *computer* thinks it's doing insert

```
PRINT CO$(VP);" ";CO$(NP):
```

at the start of line 2040.

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 that 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 (i.e. VP stays the same, only NP changes). If we're wrong about this we're actually dealing with a brand new command then having set WF to 1 ensures that a failed noun search will be followed by a verb search.

You'll have to tell me more

Aha! You thought I'd forgotten something, didn't you! (If you didn't then you've not been paying close enough attention to my pearls of wisdom.) You're quite right, though, I did promise to tell you how to deal with a situation in which the player must state not only *what* he wants to do but also *how* he's going to do it – the WHAT WITH? in my example KILL THE GOBLIN.

In fact we've already covered almost all of the coding needed – we only have to write the same thing again in a slightly different way. If you've understood the last program (Program 8.2) fully then you'll have realised that, when it comes right down to it, we're still using two-word commands to actually control the game. The difference is that we are now able to sort out, from any reasonable sentence, all the words and punctuation marks that we *don't* need. So if we want to find extra words in order to understand a command like

```
PUT THE BAG ON THE TABLE
```

in which PUT THE BAG is the main command but we also want to know *where* to put the bag, all we really need is the next *noun* in the sentence. (Of course we could be really ambitious and start analysing words like ON, IN, UNDER, etc. – but I don't want to make life too complicated at this stage.) So we already know how to look for a noun, but this time we *don't* want a verb as well – we only want to know if the player has chosen the *right* noun. To do this we simply create a *separate* subroutine which is composed of the instructions in lines 700–730 and lines 2000–2020, plus a couple of extra lines. The completed routine should look something like this:

```
1 REM *****  Extra Noun Finder  *****
2 :
3 :
29997 REM *** Are there any more
29998 REM     words in the command?
29999 :
30000 AN$="":FF=0:W=W+1:IF W>X THEN RETURN
30007 :
30008 REM *** If yes, 1st search loop
30009 :
30010 FOR NS=VT+1 TO NT
```

```
30020 IF A$(W)=CO$(NS) THEN AN$=A$(W):
      NS=NT::FF=1
30030 IF A$(W)=RIGHT$(CO$(NS),LEN(A$(W)))
      THEN B$=A$(W-1)+" "+A$(W):NS=NT:OU=1
30040 NEXT:IF OU=1 THEN OU=0:GOTO 30100
      ELSE IF FF=1 THEN RETURN
30050 IF W<X AND A$(W)<>"." THEN 30000
30060 RETURN
30097 :
30098 REM *** 2nd loop, for longer nouns
30099 :
30100 FOR QS=VT+1 TO NT
30110 IF B$=CO$(QS) THEN AN$=B$:FF=1:
      QS=NT
30120 NEXT:RETURN
```

We really don't need a complete line-by-line analysis of this routine as so much of it has already been covered in the line notes to Program 8.2. The main points to look out for here are as follows:

Line 30000: Make sure that you 'clear' or initialise AN$ at the start of the routine, and that you do move on to the next word in the input array (W=W+1) as you search for the next noun. (**Note:** This line is also used as an exit from the routine by line 30050.) In this presentation of the routine I've assumed that we are using GOSUB to access the routine. In fact you could, of course, make it a PROCedure and pass the particular noun that you are looking for across. A$(W) would then be compared with your word rather than the nouns in the CO$() array – one way of extending the list of valid nouns without adding to the noun array. If you take up this last idea, though, remember that there may be times when the player could try to use your particular word and be told that it *wasn't* a valid noun. To get round this you'll have to change line 760 of Program 8.2 to something like:

```
760 PRINT: PRINT"I CAN'T FIND A VALID NOUN
    FOR THIS SITUATION!": GOTO 500
```

Lines 30020, 30060, etc.: Notice the way that we transfer a *satisfactory* noun into another variable (AN$), if found, and then set FF to 1 for 'not found'. In this way, when we RETURN from the routine we can check whether the word(s) now in AN$ are what we are looking for with lines like:

```
9000 IF FF<>1 THEN PRINT: PRINT"HOW DO I
     "DO THAT?": GOTO 500
9010 IF AN$<>"THE SWORD" THEN PRINT:
     PRINT"PLAYING WITH "AN$" DOESN'T
     SEEM TO HELP!": GOTO 500
9020 ROUTINE TO HANDLE CORRECT COMMAND
```

Line 30050: Lastly, don't forget to keep hunting through the A$() array until you get to the end, if necessary. According to our 'rule' an invalid command erases the whole of the current command input. In this particular case we only cut the operation short if we find a full stop in A$(W), which tells us that we have *definitely* come to the end of a particular command.

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 enough 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 call to mind *The Hobbit* and *Valhalla*, which give not only word lists in their instruction booklets but also 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 and more words 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 than to find 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 that 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 to conceal 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 Ten
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 that 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, especially LOCO' BASIC (even allowing for the fact 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. The first step in this process is to look at the difference between the two fundamental methods of designing a program and turning it into finished code.

Top down or bottom up?

Despite the jokey title this section is actually worth very serious consideration. If we think about the task of preparing a computer program – and I mean *any* program, in *any* language – there are two basic approaches that we could adopt.

In the first method, known as 'Top Down' design, we would start off by writing out a very general description of the program we wanted to produce. We could then begin to write more and more detailed descriptions of the various sections, or subroutines, within the program until eventually we had worked out a set of descriptions so detailed that they could be turned directly into computer code. In other words, we would start at 'the top', and work our way *down* to the smallest details. The main advantage of this approach is that, whatever stage we are at in planning our program, we always have a pretty good idea of how all the parts are going to fit together in the end. We don't run much risk of finding ourselves with a set of routines which each work perfectly on their own but somehow don't seem to fit together to form a single whole.

But don't get me wrong – I'm not trying to say that the other method the 'Bottom Up' approach doesn't work. This second system, which involves starting out with a set of ideas, each of which is 'built up' into a

module which will fit together with the other modules to form a complete program, can work. And it can work very well – but only if you're either very lucky or very experienced at writing computer code. The main problem involved in using the Bottom Up system is that you either have to keep very detailed documentation as you go along, or be able to keep a clear picture of the *final* program in your mind even when you're dealing with the details of each of the subsections of the program. As I said, it's possible – but it certainly isn't easy. Indeed, I suspect that many of the people who buy a computer and end up selling it again because their programs never seem to work find themselves in that position because in the absence of proper instruction they have tried to use the Bottom Up approach.

I intend, therefore, to restrict myself to a description of the Top Down method of program design and writing. And the Top Down method starts like this ...

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 between the two situations 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*). (I should perhaps point out that although the programs in this book are set out as modules they started life as separate units in a block diagram.)

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 simple diagram (Fig. 10.1) and you'll see what I mean. (**Note:** in this example I've assumed that the Introduction to the main 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 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 adventure

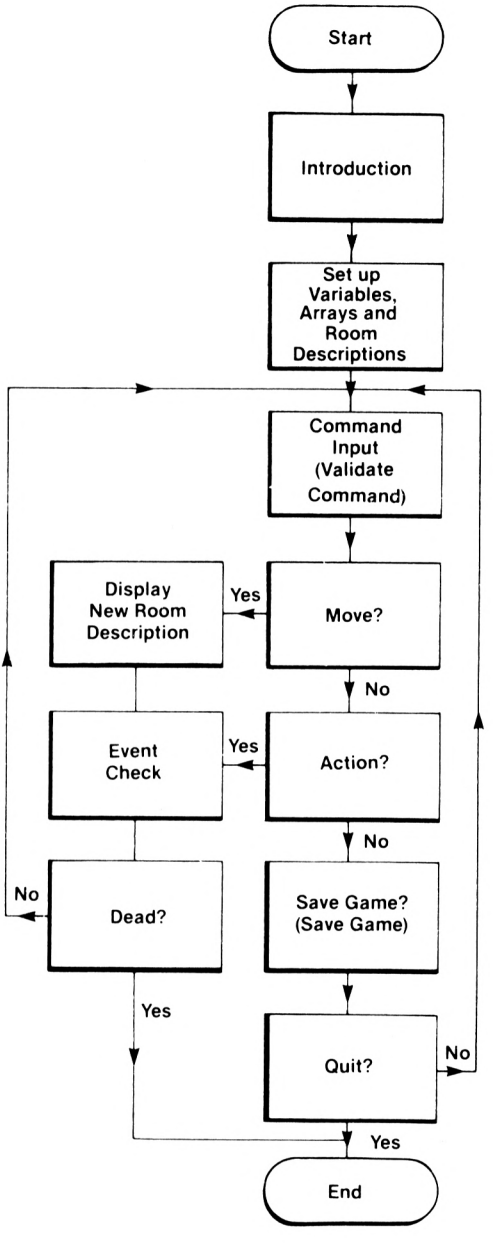


Fig. 10.1. Simple block diagram for an adventure program.

game. Which is exactly how it should be, for while the Flow Chart for each game will tend to be unique to a greater or lesser extent (and may need substantial alteration before it reaches its final form), the block

diagram is strictly 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 operations 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 being 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 debugging each section of the program much easier, but 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. 10.2).

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* that 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. Firstly 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 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

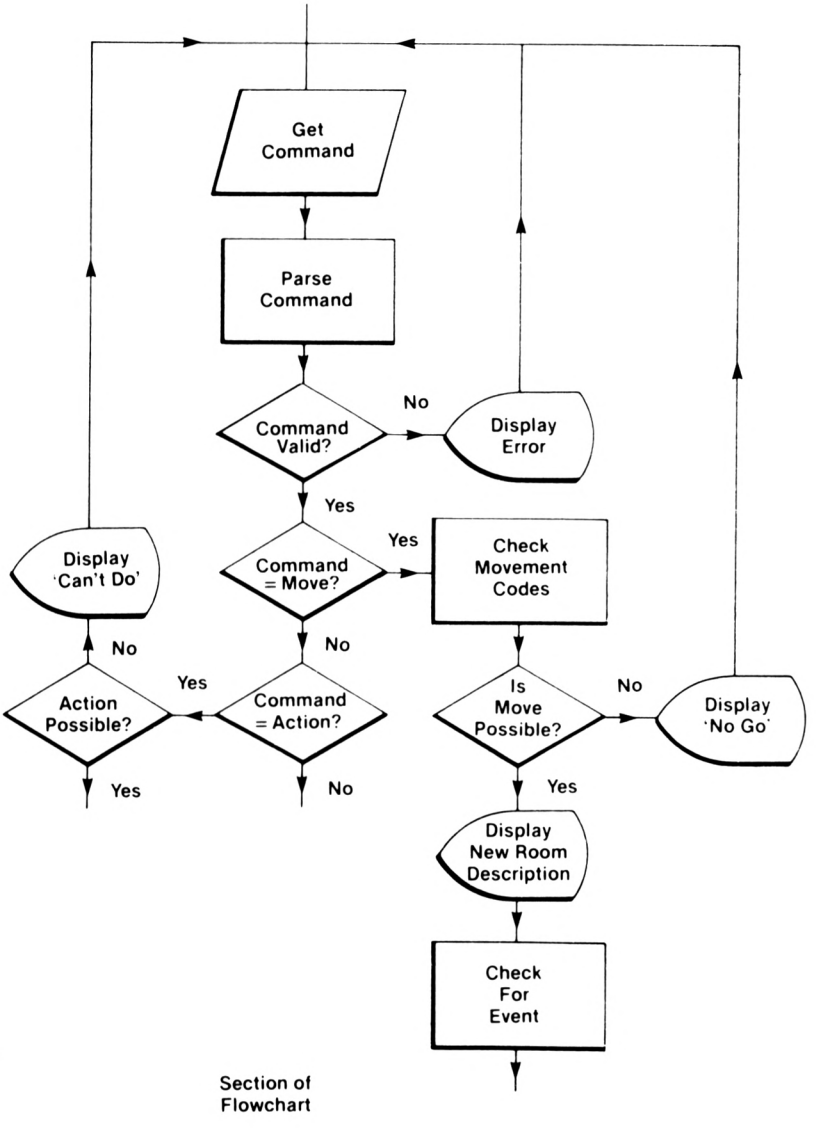


Fig. 10.2. Flowchart for one section of an adventure program.

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, we now come to the subject of

variables of which, 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. This can be quite useful, but it is not one of the facilities offered in LOCO' BASIC. It is therefore, 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. So having got that off my chest let's get back to the plot.

You may be wondering, at this point, why we would want to bother with local variables in the first place. After all, Amstrad BASIC allows us such a fantastically large range of variable names – because it takes account of *every* letter in the variable name, not just the first two as is the case in most other versions of BASIC – so that we need never run out of unique variable names however complicated our program may be. But there is a catch to all this. You must remember that if LOCO' BASIC takes account of every letter in a variable name then it must also *store* every letter of each variable name – at the cost of one byte of memory for every extra character! So while using long variable names may make it much easier to understand the program at a later date, if you want to fit long programs into memory – most adventure games are, comparatively speaking, *very* long – you will want to keep your variable names as short as possible to save program space and variable list space. So wherever it is possible, use the same variable names not once but twice, or maybe even three or four times over. We can do that by setting aside a group of variables especially for the purpose – as I have used X to control most loops and AN$ or Z$ to collect simple answers to INKEY$ input.

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 it is 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 to you 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 is 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 which included no less than twenty-eight occasions where REM lines – deleted before publication – have been used as target addresses! No doubt many readers had spotted the errors while entering the program 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 to keep a copy of the listing for 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 easily be recognised and deleted from the final version.

(2) Game size

This may seem a rather odd item to include in a discussion of programming techniques, but I would like to take a moment to look at what constitutes a 'reasonable' size for an adventure.

Several software houses have made a good deal of the fact that they can produce adventures with many hundreds, even thousands, of 'rooms' (one adventure, it is claimed, has 7000 – seven *thousand* – rooms!). So does this mean that relatively small adventures offer less value for money? My own response would be a very definite NO. In fact most of the best-selling adventures are fairly small. And quite a few of them (particularly the Scott Adams games) are positively tiny.

As I've already mentioned elsewhere in this book, one of the most important features of any adventure is the ingenuity that goes into the problem setting. Thus, whilst several Scott Adams adventures have less than fifty rooms, they still score highly with many adventure fans who appreciate the quality of the puzzles offered in each game.

If we move significantly 'up market', to the Infocom range, we find that the same rule holds true – 'small is beautiful'. All three of the ZORK adventures have only 290 rooms between them, and only one has more than 100 rooms. But check the complexity of the games! In one case you must, to move into the last part of the game, open a certain door. Easy? Not on your life! To open the door you need two items. To get just one of these items (we'll call it item X) you must find item A in one room, take it to another room, use it correctly, carry it to another room, solve a riddle, move on to another room, carry out two operations, one involving item A, move to another room, get items B, C and D, use item B correctly, move to another room, use item C correctly so as to collect item X, use item D correctly, retrace your steps – using item A yet again – and finally find the room with the door and use item X correctly.

And that's only one problem. To get the second item needed to open the door you'll have to... but no, I think you've got the point! Which brings me to the point of this section – the best use for RAM space, once you have a satisfactory set of room descriptions, is to handle a large variety of situations and problems. Without this attention to programming no amount of rooms will produce a good adventure.

(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 (preceeded 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 preferences 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 Amstrad (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 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 will then 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 beginning 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 can be found in an 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) Use integer variables (such as A% or VC%), but do remember that the Amstrad treats integer variables in a rather unusual manner (see page 37).

(c) 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 all by itself.

```
10 FOR X = 1 TO 10
20 FOR Y = 1 TO 4
30 PRINT X,Y
40 NEXT X
50 NEXT Y
```

Try RUNning this little routine as it stands and see what happens. Compare that result with what you think *should* have happened, and then change lines 40 and 50 to:

```
40 NEXT Y
50 NEXT X
```

Finally, remove the variable names from lines 40 and 50 altogether and RUN the routine again. Try to spot the difference between the second and the third sets of results – I don't think you'll find one unless you make the top values for X and Y much, much larger – then the third version should run noticeably faster.

(d) Although the Amstrad can, as we said before, read variable names of any reasonable length, it does take *time* to read extra letters (even allowing for the helpful way that variable names are arranged alphabetically and with an index). So 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.

(e) 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 program 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 Eleven
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 final chapter, that the time for this event is still several years in the future. In the meantime it's worth asking to what extent we can introduce sound and graphics into adventures written for the Amstrad *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 in going beyond pure text. 'Hang about,' says the voice. (I wondered where he'd got to!) 'What about games like *The Hobbit*!'

Which brings us back to the all-important question: how much space can we really afford to use up for sound and graphics routines? 'A good example of the limitations of a microcomputer', I answer. For despite its fairly high standard of graphics these were introduced at the price of real depth and complexity in the adventure itself. And in the BBC version the graphics had to be left out anyway because of lack of RAM space.

My own feeling is that in a BASIC adventure any graphical element needs to be economical, extremely relevant *and* well designed before it is even worth thinking about. For while *good* graphics may help to enhance a well-written game, even the best graphics that the Amstrad can manage 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, it 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 indeed 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 both graphics and sound, though well catered for in terms of hardware and all the necessary

BASIC commands, are not simple to use. I'm not saying that you've got to be some kind of programming wizard before you can produce worthwhile results, but it does take practice and experience as well as knowledge to get the *best* results that the Amstrad is capable of. Rather than try to do them justice in a few short pages, therefore, I've chosen to limit the discussion in this chapter to relatively brief descriptions of *some* of the routines you might want to use in an adventure game.

Program 11.1: Look in any window

The first thing I want to look at is a feature of the Amstrad CPC464 which I haven't seen offered in such an easy-to-handle manner on any other micro, including the BBC and the Electron. I'm referring, of course, to the way that you can constantly define and redefine text graphics 'windows' on the display screen.

The basic format for the *text* window is:

```
WINDOW #(stream number),1,2,3,4
```

where 1 represents *the column* number where the window will start and 3 represents the *row* number in which it will start. Values 2 and 4 represent the *column* and *row* respectively in which the text window will end.

Once a text window has been defined it is the *only* area of the screen which will be affected by any PRINT commands for the same stream, until the window is re-defined. Even the TAB(X) and LOCATE X,Y commands will be interpreted by the computer in relation to specific windows. Thus where TAB(10) would normally cause printing to start in the 10th column from the left, if you've created a window which *starts* in column 10 then this becomes the new column 1, and TAB(10) will cause printing to start in the 20th column from the left of the screen. Everything *outside* of the window will, of course, be unaffected by what is going on *inside* the window.

To see a very simple demonstration of how the text window works – and how to get quite an interesting colour display with the minimum of effort – try RUNning the following routine. Do read through this listing before you enter it – you'll see that several lines can be COPYed, with a new line number, to save time and effort.

```
1 REM *****  The 'Greetings Box'  *****
2 :
3 :
```

```
8 REM ***  Set inks, mode & 1st screen
9 :
10 LF=0:INK 1,3:INK 2,24:INK 3,0:MODE 1
20 PAPER #0,1:CLS
27 :
28 REM ***  Set 2nd screen
29 :
30 WINDOW #1,10,28,8,23
40 PAPER #1,2:CLS #1
47 :
48 REM ***  Repeat for 3rd screen
49 :
50 WINDOW #2,13,25,10,20
60 PAPER #2,1:CLS #2
67 :
68 REM ***  Set text colour
69 :
70 PEN #2,3
77 :
78 REM ***  Print greeting
79 :
80 LOCATE #2,5,4:PRINT #2,"HELLO":PRINT
#2, TAB(4)"THERE!!"
87 :
88 REM ***  5 second delay
89 :
90 T=TIME+1500
100 IF TIME<T THEN 100
107 :
108 REM ***  Scroll message 20 times
109 :
110 PRINT #2, CHR$(10);CHR$(10);TAB(5)"H
ELLO":PRINT #2, TAB(4)"THERE!!";CHR$(10)
120 LF=LF+1
130 IF LF<21 THEN 110
140 WINDOW #2,13,25,1,20:LOCATE #2,1,18
150 PRINT #2, TAB(5)"HELLO":PRINT #2, TA
B(4)"THERE!!";CHR$(10)
160 GOTO 150
```

Program 11.1. Graphics routine using multiple screen 'windows'.

Line-by-line analysis

Lines 10–20: Reset the mode for the computer, alter the background colour to red, and then clear the screen to that colour. The variable LF, initialised here to 0, will be used to control a 'manual loop' in lines 100–120. We also redefine our 'palette' of inks.

Lines 30–40: Reduce the size of the screen, reset the background colour to yellow, and clear the *new* screen to that colour. Notice that the outer screen is *not* affected by this command.

Lines 50–60: Repeat this process in a new screen using red once again to create the effect of a red window in a yellow square set roughly in the middle of the screen.

Lines 70–80: Next we set the *text* colour to black and PRINT a message more or less in the centre of our smallest screen.

Lines 90–130: After a five second delay the message begins to scroll within the inner window for a total of twenty passes (controlled by the value of LF). Incidentally, the size of the inner text window seems to have an effect on the rate at which the text scrolls – the smaller the window, the faster the scroll.

Line 140: Just to make this demonstration a little more interesting we now reset the size of the inner screen once again, though we only alter the *last but one* value of the window.

Lines 150–160: We then resume scrolling the greetings message – and see what happens on the screen! (I won't spoil the surprise by telling you what happens except to say that the message now scrolls to a new position on the screen.)

Program 11.2: The moving finger writes ... and draws!

Having (I hope) suitably impressed you with the simplicity of this first routine, I want to move on next to yet another of the Amstrad's extremely useful features – 'User Defined Characters'.

The first thing to be noted here is that you could redefine *any* character on the Amstrad with an ASCII code of 32 or greater – in other words, anything except the 'control' characters (see Appendix III of the User's Manual). Unfortunately from our point of view, extra memory is taken up if we want to redefine any character outside the range 240–255. That means that we can only redefine characters

without using more space if we give up some of the characters we were using for code bytes in the text packing routines in Chapter 8.

Well, let's suppose that we decide to sacrifice just a few code bytes, say a dozen or less, in order to give the adventure itself a little extra interest. How can we put those bytes to good use?

In the routine which follows I've redefined just five bytes for use in a murder mystery adventure. I won't claim that the results are exactly startling ... but see what you think.

```
1 REM *****  REDEFINED CHARACTERS  *****
2 '
3 '
10 SYMBOL AFTER 250
20 SYMBOL 250,&1F,&11,&11,&11,&1F,&20,&40,&
80
30 SYMBOL 251,&38,&38,&38,&FE,&10,&10,&10,&
10
40 SYMBOL 252,&0,&0,&30,&48,&84,&4B,&30,&0
50 SYMBOL 253,&0,&0,&C,&12,&21,&D2,&C,&0
60 SYMBOL 254,&0,&0,&3F,&68,&F0,&E0,&E0,&E0
66 '
67 REM *** PUT GRAPHICS CHARACTERS IN
68 REM *** AN ARRAY WITH THEIR NAMES
69 '
70 DIM OBJ$(4)
80 FOR X=1 TO 4
90 READ A,B,C$
100 IF X=3 THEN OBJ$(X)=CHR$(A)+CHR$(B):GOT
O 120
110 OBJ$(X)=CHR$(A)
120 OBJ$(X)=OBJ$(X)+"   "+C$
130 NEXT
137 '
138 REM *** DISPLAY OBJ$( ) ARRAY
139 '
140 FOR X=1 TO 4
150 PRINT:PRINT"You have found: "OBJ$(X)
160 NEXT
170 PRINT:END
197 '
198 REM *** OBJ$( ) DATA
199 '
200 DATA 250,32,THE MAGNIFYING GLASS,251,32
,THE KNIFE,252,253,THE HANDCUFFS,254,32,THE
 GUN
```

Program 11.2. Demonstration of user-defined graphics.

Line-by-line analysis

Lines 10–60: The information to create, with the SYMBOL AFTER command, five user-defined text characters. Just what these numbers mean will be explained in a moment. Notice they *must* be 'hex' values.

Lines 70–130: After setting up a small string array (OBJ$()) we use a simple, four-pass loop to fill it with the 'pictures' and names of *four* objects (see line 200). There are only four objects because it actually requires *two* characters – 252 and 253 – to give a reasonable representation of a pair of handcuffs. Thus line 100 picks out the third pass of the loop to create a two-character string and jump over line 110 – used during each of the other three passes – to line 120, where the name of the object (plus two blank spaces) is added on to the end of the current value of OBJ$().

Lines 140–170: Now all is prepared we can print out the four objects as they might appear in an actual adventure game.

Line 200: Finally we have the DATA to be read as values for A, B and C$ in line 90. Notice the format for each set of three items of data is FIRST CHARACTER ASCII CODE, SECOND CHARACTER ASCII CODE (32, for blank space, except in the case of the handcuffs), OBJECT NAME. Obviously if you are using single character graphics then only one numerical and one string value need be READ in line 90, and there will only be two items in each set in the DATA statement.

What we also need to know, of course, is how to prepare a 'character definition' diagram so that we can compile the right set of DATA to include in our program. This is a very simple process, thanks to the way that LOCOMOTIVE have set up the character redefinition instructions, and the only aspect that might present any problems at all is numbering the 'character grid' correctly. In Figs. 11.1 and 11.2 you will find first the basic character grid outline, and then two grids showing the diagrams from which I coded the magnifying glass and the gun used in the program.

Incidentally, these grids illustrate the make-up of a byte, something I promised to deal with in a little more detail in Chapter 7. In fact the character definition grids actually represent eight bytes stacked one on top of the other. The numbering system in Fig. 11.1 represents the value of each 'bit' within the byte. This can best be understood if you imagine that each location, or 'address', within the Amstrad consists of eight tiny switches, each of which can be either 'on' or 'off'. When the computer needs to collect a value from a particular location it 'reads' all eight bits to see which are on and which are off. Each switch that is off is

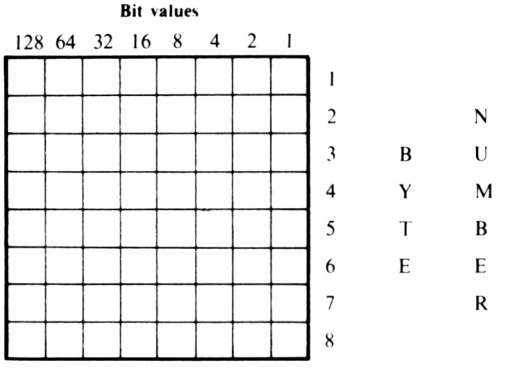


Fig. 11.1. Basic grid for user-defined characters.

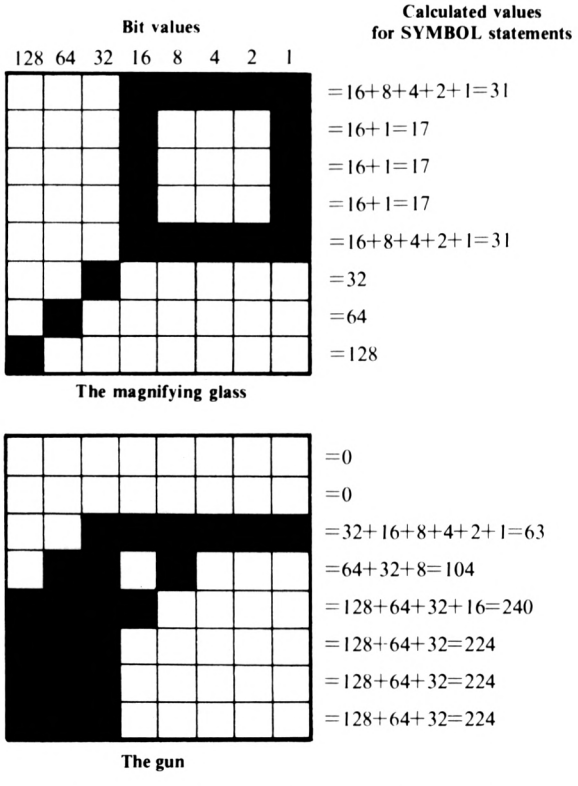


Fig. 11.2. Completed grids for two of the user-defined characters in Program 11.2.

registered by the computer as a zero. The switches that are on, however, have a numerical value according to their place in the byte. As you can see, if we work from right to left each switch has double the value of the switch or bit to its right and the highest possible value of any byte is

twice the value of the highest bit (the leftmost bit) minus 1 – that is, 128*2–1=255, or &FF. When you come to set out the values to define a text character you should use exactly the same method of calculation as that described above, given that the first byte is at the *top* of the character and the last byte is at the *bottom*. It is worth remembering, if you want to get involved with machine code, etc., that the bits are numbered from 0 to 7 (from right to left), and *not* from 1 to 8.

Sounds good

Well now, the sort of operations that I've described above are obviously pretty simple to describe and to use. You have only to draw and then code the data correctly in order to get exactly what you expect. When using the sound routines, however, getting the right collection of notes is only the start of your problems. Certainly the sound handling commands themselves are fairly straightforward, but the only way to find out what sort of sound you will produce is by old-fashioned 'trial and error'. So while experimenting with the Amstrad's sound facilities can be great fun, I think that 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), 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.

Home, home on the range

One of the few important criticisms one might make of the Amstrad User Guide is that the details of the legal values for the parameters of the various sound commands are too spread out (over five pages in Chapter 6 and three pages of Chapter 8, to be exact). For this reason I have gathered all three commands, with their parameters and values, on just a couple of pages, for easy reference.

SOUND

Command form:

SOUND CS, TP [,DUR,VOL,VE,TE,NP]

The SOUND command will operate even if values are only assigned to the first two parameters – CS and TP. All values must be given as integers, either as numbers or as numerical variables.

Legal values:

CS – 1 to 255 inclusive.
Notes: The CS parameter is specifically *bit*-based – each of the eight bits having a specific, unalterable function.

Primary ('bit') values:

1 – send sound to Channel A – the left channel if you are using the stereo output
2 – send sound to Channel B – the right channel in stereo
4 – send sound to Channel C – both left and right channels
8 – rendezvous – or synchronise with the sound going to Channel A
16 – rendezvous/synchronise with the sound going to Channel B
32 – rendezvous/synchronise with the sound going to Channel C
64 – Cut current note and 'freeze' any waiting notes until a RELEASE command is executed or until the channel is cleared by another SOUND command
128 – Clear any waiting notes on this channel (including the one being played, if any) then play current note. The queue will, of course, be empty as soon as the current note finishes.

Combination functions are built up by adding the values of two or more bits.

To end any current sounds on Channel B (if Channel B is currently empty this will have no effect) and commence a new sound or queue then the value for CS would be 130 that is, 128+2.

TP – 0 to 4096
Apart from 0, the lower the value of TP the higher the note produced. A TP value of 0 will not produce a note as such but will produce a 'noise' if parameter NP is set.

DUR – −32768 to +32767 (Default value, set by the computer if not specified by the program/user – 20)

Values −32768 to −1 – remove the minus sign to get the number of times the Volume Envelope (ENV) is to be repeated.

Value 0 – sound duration will be dictated by the accompanying Volume Envelope. If no ENV command exists then no sound will be produced.

Values 1 to 32767 – divide by 100 to give length of note in seconds.

VOL
If no Volume Envelope is defined: 0 to 7 (default value 4).
If Volume Envelope is defined: 0 to 15 (default value 12).
The higher the value of VOL the louder the note. Value 0 produces no sound at all.

VE – 0 to 15 (Default value 0)
Volume Envelope 0 is permanently defined and cannot be altered. For practical purposes only VEs 1 to 15 will be used.

TE – 0 to 15 (Default value 0)
Like Volume Envelope 0, Tone Envelope 0 is permanently fixed and cannot be redefined. TEs 1 to 15 are available to the user.

NP – 0 to 15 (Default value 0)
If the value for NP is 0 then no noise will be produced. Above 0, the higher the number the more the 'noise' will drown out any current notes.

Remember that when more than one channel has been given a 'noise' value – even though it is by separate SOUND commands – the current noise will appear on all channels opened for noise.

VOLUME ENVELOPE

Command form:

ENV VE,SC1,SS1,PT1,...SC5,SS5,PT5

Note: For the Volume Envelope command to work you only need to give values to the first four parameters. Once you have started to define a Volume Envelope, the SC, SS and PT parameters *must* be valued in complete sets, so that a complete command will end on PT1, PT2, PT3, PT4 or PT5. If you fail to define a complete set of parameters you will get a Syntax Error message at 'RUN time'. If you don't define anything except VE all lower envelopes except 0 will be reset.

Legal values:

VE – 1 to 15 (no default value)
This must be the same value as the VE parameter in the SOUND to be shaped.

SC1–SC5 – 0 to 127 (no default value)
See comments on SS1–SS5 below.

SS1–SS5 – −128 to +127 (no default value)
From SS1 onwards values may be plus or minus. However, there is a natural limit for 'normal' sounds whereby the value derived from SC*SS must not move beyond + or −15. If the result exceeds these limits the result may be interesting – it certainly won't be normal.

PT1–PT5 – 0 to 255 (no default value)
Divide PT by 100 to get length of pause before next step/end of sound in seconds (remembering that 0 will be read as 256, not nil).

TONE ENVELOPE

Command form:

ENT TE,SC1,SS1,PT1...SC5,SS5,PT5

Notes: Since the Tone Envelope works in basically the same way as the Volume Envelope, so the parameters are virtually the same (but see SS and SC sections). The important difference is, of course, that where ENV alters the *volume* of a note, ENT is used to alter the *pitch* of a note previously defined in the SOUND command.

Legal values:

TE – −15 to + 15 (but not 0, no default value)
Values −15 to −1 will be read as positive values but the envelope will be repeated until the SOUND command runs out.

Values 1 to 15 will only be executed once when accessed by the SOUND command.

SC1–SC5 – 0 to 239 (no default value)
As you will notice, the range of legal values for SC is nearly twice that allowed in the ENV command. Nevertheless, the same rule concerning the SC*SS result is in effect – only this time the limits are −5 to +5.

SS1–SS5 – −128 to +127 (no default value)
The same comments apply as were mentioned for the SS section of

the ENV command, with the exceptions mentioned above.

PT1–PT5 – 0 to 255 (no default value)
See the PT section of the ENV command. Again a 0 value for PT is interpreted as 256*1/100th of a second.

To wrap the three commands up in one simple chart:

Parameter	*Legal values*
Form of command: SOUND CS,TP,DUR,VOL,VE,TE,NP	
CS	1 to 255
TP	0 to 4096
DUR	−32768 to 32767
VOL	0 to 15
VE	0 to 15
TE	0 to 15
NP	0 to 15
Form of command: ENV EN,SC1,SS1,PT1...SC5,SS5,PT5	
EN	1 to 15
SC1 to SC5	0 to 127
SS1 to SS5	−128 to +127
PT1 to PT5	0 to 255 (=1 to 256)
Form of command: ENT EN,SC1,SS1,PT1...SC5,SS5,PT5	
EN	−15 to −1 and +1 to +15
SC1 to SC5	0 to 239
SS1 to SS5	−128 to +127
PT1 to PT5	0 to 255 (=1 to 256)

The outstanding question now is: What does it all mean? This is no easy question to answer, and I'm certainly not going to attempt too

full an answer here. Since the SOUND command is, I think, reasonably clear-cut – basically you choose a note, a length and a channel to send it through – I'd like to end this chapter by discussing some of the mysteries of 'the envelope'.

Under plain wrapper

If all we use is a pure SOUND command, as described above, then what we get is a very simple noise. How, then, do we create imitations of other instruments, notes that grow and fade, 'warble' or glide smoothly from one note to the next?

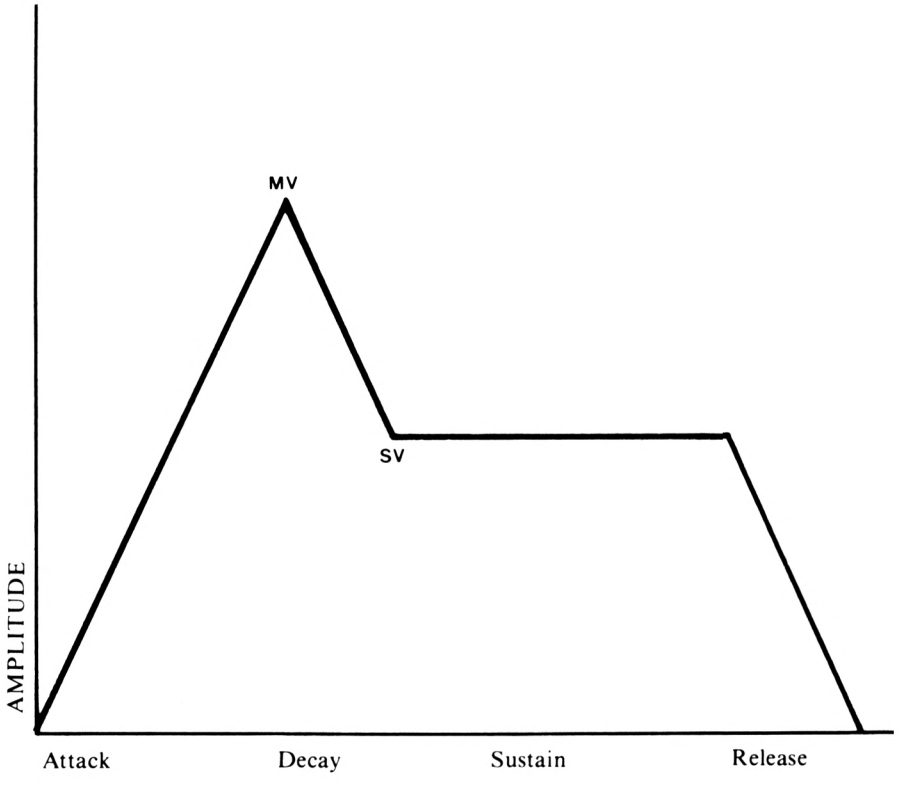


Fig. 11.3. Simple envelope.

The answer lies in the two envelope commands – ENT and ENV.

In Fig. 11.3 you will find a very basic illustration of a volume envelope. Here the volume starts at 0 and rises steadily, during the

ATTACK phase, till it reaches a peak (Maximum Volume). At this point the volume falls back somewhat during the DECAY phase to its Sustain Volume. The next stage of the envelope, the SUSTAIN phase, is usually the main part of the envelope (see Fig. 11.4), and it is usually the ATTACK and SUSTAIN phases of any envelope which make the sound of one instrument so different from another.

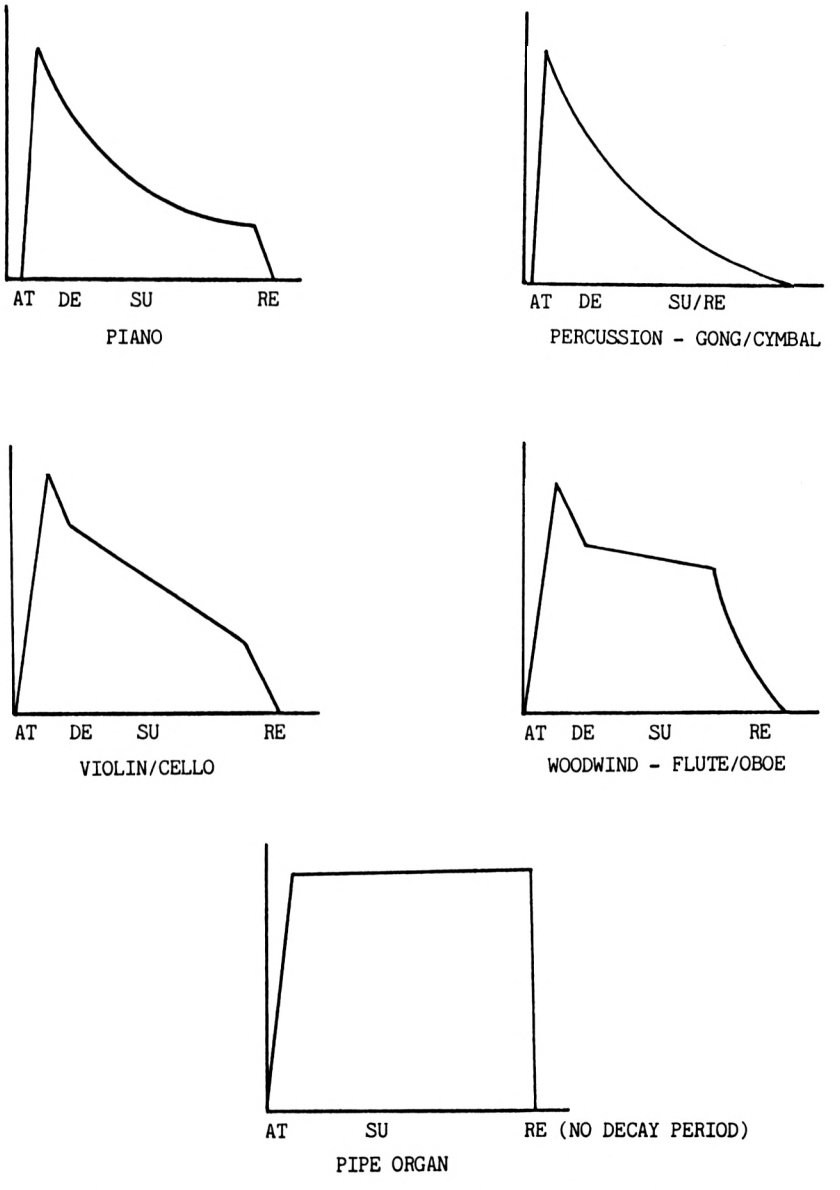


Fig. 11.4. Sound envelopes of different musical instruments.

The final stage of a standard envelope is the RELEASE phase. In theory this is the point at which the note tails off to zero volume; in practice most of the RELEASE phase is lost as the next note enters its ATTACK phase.

This explanation will, I hope, provide a clearer understanding of the way in which envelopes can affect the way that SOUND commands can be expanded upon to create far more interesting effects. Before I finish, however, there is just one last point that I'm sure will have occurred to some readers – if a standard envelope needs only three or four phases, why does the Amstrad allow five? There are, I suppose, two answers. First, if you only want three or four then you don't have to use the rest. On the other hand, if you can produce a fair imitation of 'normal' sounds with just three or four phases to the envelope – just think what you can do with five!

Chapter Twelve
What Now?

In this last chapter I want to look at some of the exciting developments that will certainly come about 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 this book has demonstrated. But it's still fun to play other people's games! So 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 will enjoy playing it as much as I enjoyed writing it.

When I first began to 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 that they had to be an 'expert'. In fact I have stuck entirely to BASIC because I felt this was the best way to show people at all levels of experience just *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 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 a 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 who 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*), the Melbourne House group, etc. My personal feeling is that the group setup is probably the best, especially for newcomers, for several reasons:

(1) It's usually easier to generate fresh ideas in a group where they 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 be tempted to 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 in the long run than a group effort where each step of the process is being discussed and reviewed.

(4) Groups cater for 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 of them only relate 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 storylines. 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 they 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 graphical 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 flowchart 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 graphical adventure the programmer and the graphical 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. Because if the game is successful and you don't market it for other machines, you can bet your life someone else will. Just look what happened to *Donkey Kong* and *Frogger!*

The method I've outlined here may seem to be concerned particularly with the production of programs for the commercial market. Not so. What I've tried to do is 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 even 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 computers 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 to 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 in '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 games 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. For 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, whilst Physics may be better suited to some kind of all-action game. Apart from the fact that it would probably be wise to stick to just one subject per adventure, 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 clear 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 adventure 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. Nevertheless, 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.

Larger 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 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, so far, been a bit of a storm in a teacup. Nevertheless, as the technology used to produce chips continues to improve it seems likely that the upgrade will eventually occur, albeit a little late in the day.

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 connects 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 be. 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), it's not difficult to imagine that graphical adventures, including animation like *Valhalla*, will become much more widespread. 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 a 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. Thus while the vocabulary and input language of adventure games is likely to go on improving for the time being, 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, I would imagine, be 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 doors!

By way of their modem connections I can imagine adventurers becoming part of nationwide link-ups wherein each player can join in, or bow out of, an ongoing game. Indeed, the idea is already 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-10 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 spent 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 with 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 'even numbered multiples 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 in 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=VP*EN+NP–EN that I used in Chapter 8. Instead each *verb* has its own area of the program reached by using VP alone to give a value for the ON GOTO command.

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 to say that the most important one will only become obvious when you try to map out the game plan. You want another clue? O.K. – it's in the middle of the game, in more senses than one. And of course there is a clue in the game itself as to what this subroutine is doing.

(3) *Legal commands*. You will notice that I have used the 'two-letter' command parsing routine in this program. In order to use it correctly please consult the list of verbs and nouns given below. Incidentally, this list is complete. There are *no* hidden commands.

All of the verbs, except for verbs A and B, should be accompanied by a 'logical' noun, or the command will be rejected.

(4) *Line length*. You will find that this listing includes one or two lines which may seem unnecessarily short. In your own games you should, of course, pack as much into each line as possible, as the start of each line – consisting of the line number and the number of bytes in the line – also takes up valuable RAM space. In fact I have broken the lines down to this extent mainly because it makes it much easier to actually type the lines into your own machine – you won't discover later that you have to edit five or six screen lines just to weed out one or two tiny mistakes. And talking of mistakes ...

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

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 beginning 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, 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 Collins directly: they are publishers, not computer experts. If, on the other hand, you would like to write to me – care of Collins – 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
2 '             THE MISSING ADVENTURE
3 :
4 '                by A.J. Bradbury
5 :
6 '                  COPYRIGHT 1984
9 :
10 :
1000 GOSUB 16000
1100 AD%=RND*9+1
1500 GOTO 5000
2000 PRINT: PRINT:PRINT"You are carrying: "
2010 FOR X=1 TO 10:IF OB$(X,1)="-1" THEN PR
INT"a "OB$(X,0)
2020 NEXT
2030 PRINT:PRINT"A total weight of "IN" lbs
."
2040 GOTO 7000
2100 SF=3
2200 IF SF<>3 OR PL<>AD% THEN 2250 ELSE PRI
NT:PRINT:PRINT TAB(10);CHR$(24)" CONGRATULA
TIONS!!! ";CHR$(24):PRINT
2210 PRINT"   You've found the missing adve
nture - Now all you need is a good software
     house to market it and make yourfortun
e for you.":PRINT:PRINT"But that's another
game altogether ...":END
2250 PRINT
2260 FOR X=2 TO 4
2270 IF VAL(OB$(X,1))=PL THEN PRINT:PRINT"Y
ou've found the "OB$(X,0)"!":O1=1
2280 NEXT
2290 IF PL<>1 AND O1=1 THEN O1=0:GOTO 7000
2300 IF PL=1 THEN PRINT"The object is a sma
```

```
ll badge with a blackfigure 6 on it.":GOTO
7000
2310 PRINT:PRINT"What you see is what there
 is!":GOTO 7000
3000 PRINT:PRINT"Well, well - the safe has
a combination lock (now where have you hear
d that      before?)"
3010 PRINT:PRINT"Would you like to try to o
pen it -":PRINT:PRINT"(Enter Y or N): ";
3020 AN$=INKEY$:IF AN$="" THEN 3020
3030 IF UPPER$(AN$)<>"Y" THEN 7000
3040 LO%=RND*900+100:LO$=STR$(LO%):LO$=RIGH
T$(LO$,3)
3050 Q=1:WHILE (MT-Q)<7 AND Q<4
3060 PRINT:PRINT"Please enter digit number"
Q": ";
3070 SA$=INKEY$:IF SA$="" THEN 3070
3080 IF SA$<>MID$(LO$,Q,1) THEN PRINT:PRINT
"Sorry - try again." ELSE PRINT:PRINT"Good
- you have"Q"out of 3 digits.":Q=Q+1
3090 MT=MT+1:WEND
3100 IF Q=4 THEN 3150
3110 PRINT:PRINT"You've guessed it - the sa
fe is booby-  trapped.  You fall through a
hatchway   into the room below!":PRINT
3120 IF OB$(7,1)="-1" THEN OB$(7,1)="4":PRI
NT"... and you've dropped the crucifix!"
3130 GOTO 11850
3150 PRINT:PRINT"Oh dear! ...":GOTO 3110
4600 HF=0:PRINT:PRINT"You can see:"
4610 FOR X=5 TO 10
4620 IF VAL(OB$(X,1))=PL THEN HF=1:PRINT"A
";OB$(X,0)
4630 NEXT
4640 IF HF=1 THEN SI$="and"
4650 IF HF=0 AND (PL=1 OR PL=3 OR PL=7) THE
N SI$="nothing except":HF=1:PRINT
4670 IF PL=1 OR PL=3 THEN PRINT:PRINT SI$"
a sign on the wall."
4680 IF PL=7 THEN PRINT SI$" a sign over th
e west door"
4690 IF HF=0 THEN PRINT"Nothing of any grea
t interest."
4700 HF=0:RETURN
5000 FOR X=1 TO 4
```

```
5010 T%=RND*6+2:IF T%=3 THEN 5010
5020 FOR Y=1 TO X:IF VAL(OB$(Y,1))=T% THEN
GOTO 5010
5030 NEXT Y
5040 OB$(X,1)=STR$(T%)
5050 NEXT X
5060 OB$(2,1)="3"
6000 PRINT:PRINT:PRINT:SH=12:PL=1:PRINT RD$
(PL):SF=0:FF=0
6010 GOSUB 4600
7000 GOSUB 9000:PRINT:PRINT"What now? ";
7010 V$=INKEY$:IF V$="" THEN 7010
7020 V$=UPPER$(V$):VP=INSTR(VE$,V$)
7030 IF VP=0 THEN PRINT:GOTO 7200
7040 PRINT V$(VP);
7050 IF VP<3 THEN ON VP GOTO 2000,2200
7080 N$=INKEY$:IF N$="" THEN 7080
7090 IF N$=CHR$(127) THEN FOR X=1 TO LEN(V$
(VP)):PRINT BA$;:NEXT:GOTO 7010
7100 N$=UPPER$(N$):NP=INSTR(NO$,N$)
7110 IF NP=0 THEN PRINT:GOTO 7210
7130 IF NP<18 THEN PRINT" the";
7140 PRINT" ";N$(NP);:T=TIME+600
7145 REM *** I have inserted a 2 second del
ay here.  To remove it DELETE everything fr
om (and including) the SECOND semi-colon in
 line 7140 through to the end of line 7160
7150 AL$=INKEY$:IF AL$="" AND TIME<T THEN 7
150
7160 IF AL$=CHR$(127) THEN FOR X=1 TO LEN(N
$(NP))+1:PRINT BA$;:NEXT:GOTO 7080
7170 PRINT" ":IF VP>6 THEN VP=VP-6:GOTO 719
0
7180 ON VP GOTO 1,1,10000,10200,10400,10600
7190 ON VP GOTO 10800,11000,11200,11400,116
00,11800
7200 PRINT:PRINT"I don't understand "V$:GOT
O 7000
7210 PRINT:PRINT"I don't understand "V$(VP)
" "N$:GOTO 7000
9000 MS=VAL(RIGHT$(MC$,1))
9010 FOR X=2 TO 6:MC(3,X)=VAL(MID$(MC$,X-1,
1)):NEXT
9020 MC(3,1)=MS:MC$="":FOR X=1 TO 6:MC$=MC$
+CHR$(MC(3,X)+48):NEXT
```

```
9100 IF OB$(1,1)="0" THEN RETURN
9110 BL=VAL(OB$(1,1)):BM%=RND*5+1:IF MC(BL,
BM%)=0 THEN RETURN
9120 BL=MC(BL,BM%):OB$(1,1)=CHR$(BL+48):IF
BL<>PL THEN RETURN
9200 PRINT:PRINT"Oops!  You've just found t
he dreaded BUG- you have no choice but to s
tand and   fight ..."
9210 FOR X=1 TO 3000:NEXT
9220 ML=2:MH=21:SL=1:IF FF=1 THEN SL=9
9230 PS%=(SL*(RND*6+1)+PT)/6
9240 MH=MH-PS%:IF MH<1 THEN 9300
9250 MS%=(ML*(RND*6+1)+MH)/6
9260 SH=SH-MS%:IF SH<1 THEN 9400
9270 GOTO 9230
9300 PRINT:PRINT"Well done - you've defeate
d the BUG.    You are free to continue.":OB
$(1,1)="0":RETURN
9400 PRINT:PRINT"What a sad end for such a
brave advent- urer.  To put it another way:
 'T-T-T-   That's all for now folks!":END
10000 NR=NP-17:IF NR<1 OR NR>6 THEN N$=N$(N
P):GOTO 7210
10010 IF MC(PL,NR)=0 THEN KF$=" ":GOTO 1005
0
10020 IF PL=7 AND NM=4 AND KF=0 THEN KF$="-
 yet!":GOTO 10050
10040 PL=MC(PL,NR):PRINT:PRINT"O.K.":PRINT:
PRINT RD$(PL):GOSUB 4600:GOTO 7000
10050 PRINT:PRINT"Sorry - you can't go that
 way"KF$:GOTO 7000
10200 IF NP>10 THEN N$=N$(NP):GOTO 7210
10210 FOR X=1 TO 10
10220 IF N$(NP)=OB$(X,0) THEN CH=X:GOTO 102
50
10230 NEXT
10240 PRINT:PRINT"Funny - I can't see "N$(N
P)" here":GOTO 7000
10250 IF IN+VAL(OB$(CH,2))>20 THEN PRINT:PR
INT"Uh, uh.  You can't carry anything that
 big - not at the moment, anyway.":GOTO 700
0
10260 IF OB$(CH,1)="-1" THEN PRINT:PRINT"Wa
key, wakey!  You already have the":PRINT N$
(NP):GOTO 7000
```

```
10270 IF OB$(CH,1)="0" THEN PRINT:PRINT"Sor
ry - the "N$(NP):PRINT"isn't available.":GO
TO 7000
10280 IF VAL(OB$(CH,1))<>PL THEN PRINT"No!
- the "N$(NP)" isn't here.":GOTO 7000
10300 PRINT:PRINT"O.K. - you now have the "
N$(NP)"."
10310 OB$(CH,1)="-1":IN=IN+VAL(OB$(CH,2)):I
F CH>2 AND CH<6 THEN SF=SF+1
10320 IF CH=2 THEN FF=1
10330 IF CH=5 THEN MC(7,4)=10
10340 GOTO 7000
10400 FOR X=1 TO 10
10410 IF OB$(X,0)=N$(NP) THEN TE=X:GOTO 104
40
10420 NEXT
10430 PRINT:PRINT"Sorry - there is no "N$(N
P):GOTO 7000
10440 Q=0
10450 IF OB$(TE,1)<>"-1" THEN PRINT:PRINT"Y
ou can't drop what you don't have!!":GOTO 7
000
10470 PRINT:PRINT"O.K.":OB$(TE,1)=STR$(PL):
IF TE>2 AND TE<6 THEN SF=SF-1
10480 IF TE=2 THEN FF=0
10490 IF TE=5 THEN MC(7,4)=0
10500 IN=IN-VAL(OB$(TE,2)):GOTO 7000
10600 IF PL<>4 AND PL<>5 AND PL<>7 THEN 107
00
10610 IF PL=5 AND N$(NO)="hatchet" THEN 116
10
10620 IF PL=4 AND N$(NP)="key" THEN PRINT:P
RINT"Oh dear - the key doesn't fit the safe
'slock.":GOTO 3000
10630 IF PL=7 AND N$(NP)="ring" THEN PRINT:
PRINT"As you hold up the ring Bulbous simpl
y  fades away like a puff of smoke.  You
are free to continue.":GOTO 7000
10700 PRINT:PRINT"Hmmm - using the "N$(NP)"
 doesn't":PRINT"seem to have any effect.":G
OTO 7000
10800 IF PL<>1 AND PL<>3 AND PL<>7 AND N$(N
P)<>"map" THEN PRINT:PRINT"I don't see anyt
hing to read here.":GOTO 7000
10810 IF N$(NP)="map" THEN PRINT:PRINT"Well
```

```
 now, how strange.  The map doesn't seem to
 apply to this adventure.":GOTO 7000 ELSE I
F N$(NP)<>"sign" THEN 11000
10820 IF PL=1 THEN S$="There's no such thin
g as a FREE lunch!"
10830 IF PL=3 THEN S$="The world keeps turn
ing - and so do I."
10840 IF PL=7 THEN S$="Abandon HOPE all ye
who enter here!!!"
10850 PRINT:PRINT"The sign says:":PRINT:PRI
NT S$:GOTO 7000
11000 IF PL<>4 THEN PRINT:PRINT"Huh?":GOTO
7000
11010 IF PL=4 AND N$(NP)="safe" THEN N$(NP)
="key":GOTO 10620
11020 N$=N$(NP):GOTO 7210
11200 IF PL<>7 THEN PRINT:PRINT"I can't - n
othing here is locked.":GOTO 7000
11210 IF N$(NP)="key" THEN PRINT:PRINT"O.K.
  But don't say you weren't warned!!":NP=20
:GOTO 10000
11230 GOTO 7000
11400 IF PL<>3 AND PL<>5 AND PL<>8 THEN PRI
NT:PRINT"There's nothing here to be climbed
!":GOTO 7000
11410 IF (PL=3 OR PL=8) AND N$(NP)="stairs"
 THEN NP=21:GOTO 10000
11420 IF PL=5 AND N$(NP)="tree" THEN PRINT:
PRINT"O.K.  Uh-oh, a branch just broke, you
   fall and break one arm!":SH=SH-3:GOTO 11
900
11440 PRINT:PRINT"No - you can't do that he
re.":GOTO 7000
11600 IF PL<>5 THEN PRINT:PRINT"There is no
thing here you can chop down.":GOTO 7000
11610 PRINT:PRINT"O.K.  You now have a pile
 of logs -     which promptly fade from sig
ht.  The    PHANTOM CONSERVATIONIST strikes
 again!":GOTO 7000
11800 IF PL<>7 AND PL<>8 THEN PRINT:PRINT"T
here's nothing here to be killed        (exc
ept YOU, of course!).":GOTO 7000
11810 IF PL=7 THEN PRINT:PRINT"Would you be
lieve it - Bulbous is pro-  tected by a mag
ic spell and cannot be   killed.  Still, yo
```

```
u've wasted strength  by your efforts.":SH=
SH-2:GOTO 11900
11830 IF PL=8 AND OB$(7,1)="-1" THEN PRINT:
PRINT"Well done!!  After one look at the
   crucifix the vampire fades into the
shadows.  You are free to continue.":GOTO 7
000
11850 PRINT:PRINT"Obviously you don't watch
 many horror   films.  Without a crucifix y
ou are       powerless against the vampire."
:SH=0
11900 IF SH<1 THEN PRINT:PRINT"Oh dear, you
r life-force (Strength) is  all gone.  R.I.
P.!!!":END
11910 GOTO 7000
13000 DIM V$(12),N$(23),MC(10,6),OB$(10,2),
RD$(10)
13010 FOR X=1 TO 12:READ V$(X):NEXT
13020 FOR X=1 TO 23:READ N$(X):NEXT
13030 FOR X=1 TO 10:FOR Y=1 TO 6:READ MC(X,
Y):NEXT:NEXT
13040 FOR X=1 TO 10:FOR Y=0 TO 2:READ OB$(X
,Y):NEXT:NEXT
13050 FOR X=1 TO 10:READ RD$(X):NEXT
13060 VE$="ABCDEFGHIJKL":NO$="ABCDEFGHIJKLM
NOPQRSTUVW"
13070 BA$=CHR$(8)+CHR$(16):MC$="624598"
13100 RETURN
14000 DATA inventory,search,go,get,drop,use
,read,open,unlock,climb,chop down,kill
14100 DATA the BUG,flowchart,map,storyline,
key,hatchet,crucifix,badge,ring,ray gun
14110 DATA stairs,tree,safe,vampire,Gobbet,
sign,door
14120 DATA north,south,east,west,up,down
14200 DATA 2,0,0,0,0,0,3,1,0,0,0,0
14210 DATA 6,2,4,5,9,8,0,0,3,0,0,0
14220 DATA 0,5,7,3,0,0,7,3,0,0,0,0
14230 DATA 0,6,5,0,0,0,0,0,0,0,3,0
14240 DATA 0,0,0,0,0,3,0,0,7,0,0,0
14300 DATA the BUG,3,0,flowchart,3,2,map,3,
2,storyline,3,2
14310 DATA key,7,1,hatchet,6,16,crucifix,4,
4
14320 DATA badge,1,1,ring,5,1,ray gun,2,6
```

```
14400 DATA You are inside Fantasia Internat
ional.  This first room is about 10 feet by
 6   with a single (barred) window high up
onthe west wall.  There is a small object o
n the floor - and what looks like a    hand
le on the far wall.
14500 DATA Well 'beam me up Snotty' - you'r
e aboardthat well-known flying cake tin the
 USS Split Infinitive.  Unfortunately there
  doesn't seem to be anyone around - so ify
ou get scared you'll have no-one to    'Kli
ng on' to!!!
14600 DATA You are in a small cavern.  The
walls   and ceiling are covered with cobweb
s.   The only light comes from two blazing
  torches.  In the dust on the floor are  s
everal sets of strange tracks (such as migh
t be made by a giant insect!).
14700 DATA "You have entered a library full
 of well-made leather furniture, rows and r
ows ofbooks - and a dead body left over fro
m amurder adventure.  A picture on the west
wall has been pulled back to reveal a   sma
ll safe.  The safe is closed."
14800 DATA You are in what seems to be an u
nending forest.  It's quite a nice forest -
 as  forests go - but that's the best that
  could be said for it.  Still - the path m
ust lead somewhere because there's a   brok
en sign to the east that says 'To Go
14900 DATA "You seem to be on a desert isla
nd -      there's nothing but sand, sea and
a few palm trees as far as the eye can see.
   Nothing much seems to be going on here
at all (though you you could try digging- i
f you have a spade!)."
15000 DATA CROOKS TOURS welcome you to Gobb
etania -the land that time forgot.  You are
      greeted by your courier (?) - a small
  figure with a large sword.  It's BulbousF
agend (who can re-incarnate at will).  But
why is he waving that sword at you?
15100 DATA As you enter a damp and gloomy c
rypt a  bat flies past you into the shadows
.  Inthe far corner the lid of a coffin ope
```

```
ns- a smiling gentleman with long teeth   a
nd an even longer black cloak steps outand
glides silently towards you.
152ØØ DATA At the top of the stairs you fin
d your- self in the cockpit of a Jumbo Jet.
  Onelook at the control panel tells you th
atthe plane is nearly out of fuel - and   g
oing into a steep dive.  Isn't this   exci
ting!!!
153ØØ DATA O.K. - here we go.  You step thr
ough thedoor into a room where a new game c
alledAPOCALYPSE YESTERDAY was being tested.
  Unfortunately something has gone wrong -t
he room is a nuclear wasteland - and   the
door has automatically re-sealed!!!
16ØØØ CLS:LOCATE 12,3:PRINT CHR$(24)" THE C
ASE OF THE "CHR$(24)
16Ø1Ø PRINT:PRINT TAB(11);CHR$(24)" MISSING
 ADVENTURE "CHR$(24)
16Ø2Ø PRINT:PRINT"  It was a dark and gloom
y day and the  rain was coming down in shee
ts - cotton sheets, satin sheets - I should
 have    stayed in bed!"
16Ø3Ø PRINT:PRINT"  I'd spent several days
trying to trackdown Eddy the Kwill, the ult
imate in    undesirable characters.  I'd en
ded up atthe wrong end of a blind alley."
16Ø4Ø PRINT:PRINT" Suddenly a door at the e
nd of the alleyopened.  A hand beckoned to
me, but to  be honest I wasn't sure what to
 do."
16Ø5Ø GOSUB 13ØØØ
16Ø6Ø PRINT:PRINT"  I think I'll leave the
choice to you. Press D and I go through the
 door.":PRINT"Press any other key and I qui
t the case."
161ØØ Z$=INKEY$:IF Z$="" THEN 161ØØ
1611Ø Z$=UPPER$(Z$):IF Z$<>"D" THEN CLS:PRI
NT:PRINT"O.K.  -  I quit!":END
1612Ø CLS:PRINT:PRINT"Good choice blue/brow
n/green/grey eyes! Now I'm you, and you're
me."
1613Ø PRINT:PRINT"Which means that from now
 on this is    your case (don't you just lo
ve surprises!!!)."
1614Ø PRINT:PRINT"By the way - Good Luck -
you're going toneed it!!":RETURN
```

Index

Page references in **bold** type denote Figures and Program listings.

CREATE YOUR OWN AMSTRAD ADVENTURES!

Users of both the Amstrad CPC464 and the CPC664 will be able, with this book, to harness the tremendous power of LOCOMOTIVE BASIC in order to create truly remarkable adventure games.

Whether you're an experienced programmer or a complete novice you will find all the information needed to prepare, map and program complete adventures.

The Author

A. J. Bradbury, who also writes under the name of John Noad, is a computer studies teacher and a keen adventurer. His articles have appeared in *Windfall* and *Personal Computer News*.

Other books for Amstrad users

AMSTRAD COMPUTING
Ian Sinclair
0 00 383120 5

SENSATIONAL GAMES FOR THE AMSTRAD CPC464
Jim Gregory
0 00 383121 3

40 EDUCATIONAL GAMES FOR THE AMSTRAD CPC464
Vince Apps
0 00 383119 1

INTRODUCING AMSTRAD MACHINE CODE
Ian Sinclair
0 00 383079 9

FILING SYSTEMS AND DATABASES FOR THE AMSTRAD CPC464
A. P. Stephenson and D. J. Stephenson
0 00 383102 7

PRACTICAL PROGRAMS FOR THE AMSTRAD CPC464
Audrey Bishop and Owen Bishop
0 00 383082 9

THE AMSTRAD CPC464 DISC SYSTEM
Ian Sinclair
0 00 383177 9

Front cover illustration by Colin Hay

COLLINS
Printed in Great Britain

ISBN 0-00-383078-0

£7.95 net

BRADBURY **ADVENTURE GAMES FOR THE AMSTRAD CPC464** COLLINS

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/