

GLENTOP

COMPUTER BOOKS

PROGRAMMING

in

C

on the

AMSTRAD

464-664-6128

IAN SINCLAIR

Programming in C on the AMSTRAD

GLENTOP[□]
PUBLISHERS □ LIMITED

Programming

in

C

on the

AMSTRAD

Ian Sinclair

JANUARY 1986

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publishers for any errors or omissions contained herein.

COPYRIGHT © Ian Sinclair 1986
World rights reserved.

No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including but not limited to photography, photocopy, magnetic or other recording means, without prior permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

ISBN 0 907792 86 3

Published by:

Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: 01-441-4130

Contents

Preface

Chapter 1: Introduction to 'C'

Why 'C'? ● The order of things ● Program structure ● Source code and Machine code ● HiSoft C summarised

Chapter 2: Starting 'C' programs

First steps ● Output to screen ● Using constants ● Constants ● More variables ● Getting a value ● Other variable types

Chapter 3: Functions and the Library

Functions ● The Library ● Planning ● Some operations ● More planning ● More functions

Chapter 4: Pointers

Arrays ● Strings at last ● Arrays of strings

Chapter 5: Menus, choices and files

Other cases ● Recording data ● String files

Chapter 6: More structured types

Filing structures ● Reading back structures ● Working with structure files ● Sorting a file ● Record nests and choices

Chapter 7: More about pointers

Arrays of pointers ● Pointers to functions ● Linked lists

Chapter 8: String functions

Left, right and middle ● Concatenation and insertion ● Other string and character functions

Chapter 9: Graphics, sound and BASIC conversion

Sounds unlimited ● Using envelopes

Chapter 10: Assortment

Hints & tips ● Other actions ● More complications ● The # commands ● Statics in functions ● Binary and machine code ● Inserting machine code

Appendix A: Binary, Octal and Hex Codes

Index

Preface

Though 'C' has been used for a considerable time as a language for writing compilers, word-processors and games, it has never become as well known as it ought to be, and has certainly never displaced BASIC as a language for microcomputers. This has been due to several reasons, some of which are no longer valid. One reason has been that BASIC, because it allowed direct keyboard commands, was an easier language to learn. This is certainly true, but it is also true that it is easier to learn to program badly in BASIC, even in Locomotive BASIC, than to program well. Another restriction was that many 'C' compilers did not produce machine code which could then be recorded and used on any suitable machine. It was common to find that 'C' programs could be run only on machines which were equipped with a complete 'C' compiler installed. The most vexing problem was that 'C' was simply not available for machines with small memory sizes. The result was that this useful, fascinating, though sometimes **infuriating** language was confined to programmers who had mini rather than micro-computers.

The arrival of HiSoft C has changed the outlook completely. The version around which this book is written has been designed for the Amstrad machines, from the CPC464 upwards, including the CPC664 and CPC6128. This version of 'C' supports direct commands which, though not as flexible and simple as those of BASIC, do at least allow to escape some of the frustration of using other versions of 'C'. A forthcoming version of 'C' which runs under CP/M should be even more useful. Now, using HiSoft C, the programmer can operate with a high level compiled language which includes instructions for the high resolution graphics and sound, and which compiles to very fast-running machine code. The serious programmer who uses discs can also produce 'C' programs in machine code which will run on Amstrad machines that **are not equipped with HiSoft C**. This language, then, provides all that is lacking in BASIC, and much more.

I have assumed that the reader of this book will have programmed in BASIC. Since the book is written for the Amstrad user, this seems to be a reasonable assumption, and it provides a basis of comparison between Locomotive BASIC and 'C'. In addition, I have assumed that the reader who wishes to use graphics and sound is familiar with the methods of Locomotive BASIC. This is important, because in HiSoft C, the graphics routines are rather similar, and the simpler types of sound instructions should be useable by anyone who has used the sound system in BASIC. HiSoft C, however, offers significant extensions to these commands. These extensions form part of the HiSoft Library of 'C' functions, and very many of these functions have been more fully described in this book. In addition, some of the non-standard features of HiSoft C, such as the **inline** statement, have been explained, with examples, in a more complete way that has been provided in the manual.

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

Because the way that you write 'C' programs is very closely tied to the way that you design a program, I have linked these topics together rather than trying to deal with the design of programs separately. The book has been written entirely around the conventional 'top-down' programming method, and the programming has been approached in the same way. The system which was used was a CPC464 with a disc drive, but the book is equally suitable for the user who has bought HiSoft C on cassette. Obviously, the book will also be suitable for the buyers of the CPC6128 model which are now available. Throughout the book, the carriage-return/newline key has been referred to as (RETURN) (as on the CPC6128) or as (ENTER) (as on the CPC464). Where I have used commands that are peculiar to the disc system, I have pointed this out. My HiSoft C disc was a stock item, purchased by mail, and not a pre-production sample. The version number was V1.2. The reader should find, then, that there are no significant differences between the 'C' of my descriptions and the 'C' that he/she has bought. The programs that appear in this book have been photo-copied from listings which were produced by an EPSON printer reading the 'C' program direct from disc. The listings in the book are therefore **exactly** of the programs which I used. No line numbers are shown, because the line numbers that appear on the screen are not sent to the printer.

Finally, I am very grateful to HiSoft for this excellent version of 'C', which has breathed life into the learning of this fascinating language. I am indebted to them for permission to print a few of the large number of library functions which form part of the HiSoft C package for the Amstrad. In particular, I want to thank Dave Howarth for many interesting hints. I am also most grateful to Dr. Peter Holmes, of Glentop Publishers, who warmed to the prospects of 'C', and commissioned this book.

Ian R. Sinclair

September 1985

Chapter 1

Introduction to 'C'.

Why 'C'?

If you program in BASIC and find the language perfectly adequate, it's reasonable to ask "why use 'C' ?". Certainly, if you find the Locomotive BASIC of the Amstrad completely suited for what you want to do, you probably don't need 'C'. On the other hand, it's much more likely that you have found a lot of items for which BASIC, even the excellent Locomotive BASIC, is not well suited. One obvious set of such items on every version of BASIC is the programming of database records. Others are less obvious, unless you are trying to write business software that requires the use of lists, or games software which uses sets of items. BASIC can cope with these needs only in roundabout ways, and if you have only ever programmed in BASIC, you simply don't realise how much effort you are having to make for what are quite simple actions in other languages.

Another reason for using 'C', of course, is speed. BASIC is an interpreted language. This means that every instruction in a BASIC program is carried out as the machine comes to it. A computer carries out these instructions by converting them into machine code and running the machine code. An interpreter takes the instructions one by one, finds the machine code for an instruction, runs that instruction, then looks for the next one. By contrast, a compiler deals with the whole of a program and converts it into one piece of machine code. This code can then be recorded and run. The advantage of this is that all of the looking-up and converting steps are done once, in the action which is called **compiling**, and this compiled code can run fast, usually almost as fast as machine code which has been written using an Assembler. When you use an interpreted language, the interpretation has to be done for each step and for each time the program is run. For example, suppose that your program consists of a loop which prints a string variable value fifty times. An interpreter would look for the machine code for the print operation, find the address and length of the string of characters, and carry out the same action fifty times. All of the looking up is repeated **on each pass** through the loop. A compiler, by contrast, would generate machine code which carried out the print action in a loop. This generation would be done once, during compiling, and because the looking up actions do not have to be done again, the result is very much faster. The speed of a compiled language may be important for you if you want to write either business or games software, though the advantage is less for some types of games programs. The reason is that the 'C' compiler makes use of the built-in screen routines of the Amstrad machines, which are comparatively slow. HiSoft C, however, allows you to mix in machine code with your 'C' so as to carry out actions like direct access to the screen memory.

Yet another reason for using 'C' is that the language is much closer to the languages that are used on larger machines. It is also 'portable' in the sense that a program which is written in 'C' for one machine will probably work with another machine. This assumes only that there are no instructions which are peculiar to one machine, and 'C' is particularly good in this respect. The way that 'C' is designed, there are a few instructions that you find in all versions of 'C', and a lot of actions (the library functions) that you can call up to make any particular machine go through its tricks. Above all, though, 'C' is a language that forces you to think carefully about what you want a program to do. It is possible to write poorly-designed and sloppy programs in 'C'. It is also possible to write neat and tidy programs in BASIC. In the normal run of events, however, both of these are unusual. One of the things that endears 'C' to old BASIC programmers is that 'C' is a flexible language. You aren't tied to the very rigid rules that you find, for example, in Pascal. On the other hand, you pay for this flexibility in being allowed (some might say encouraged) to make mistakes. The 'C' compiler won't stop you when you try to do something silly, and it can sometimes be very difficult to find out just what is causing the trouble.

The order of things.

'C' is not simply a compiled language, it is a language in which programs can be written which compile to unusually compact and fast machine code. This is possible only if you program in the correct order, though. The most important idea to get used to, if your programming experience is in BASIC, is that types of variables and their names have to be **declared** before they can be used. In BASIC, you can write a line like:

```
100 A%=5
```

which introduces a variable name, A%, with the % sign meaning that this is an integer. At the same time, the value to which A% is assigned is made equal to 5. A 'C' programmer can write a very similar kind of line, but it has to appear early in a program, before the variable will be used. The 'C' form of this line will be:

```
static int a=5;
```

with **int** used instead of the % sign to mean that **a** represents an integer. The word **static** refers to the way that the value is stored, and will be explained later. Using this **declaration** means that we use only **a** in the program, not **int a** or **a%**. The use of the equality sign and the 5 then assigns a value of 5 to **a**. The assignment does *not* have to be made here, it can be done later in the program. Declaring a variable type, and assigning a value in one step is just one of the short cuts that 'C' allows and which makes it such a very interesting and challenging language.

This idea of declaring what type of variable is represented is not one that appears much in BASIC. The Amstrad machines, along with a few machines which use Microsoft BASIC (such as the MSX machines) use instructions such as DEFINT. This allows you to define how letters will be used in the program. For example, DEFINT A-D in Microsoft BASIC means that any variable name which starts with the letters A, B, C, or D will be an integer variable. This means that you no longer have to mark integer variables, like A%, BY%, COWS% and so on, to show that they **are** integers. The DEFINT statement at the start of the program has done this for you, and it's also possible to define string or real-number variables in a similar way.

In 'C', however, this idea of declaring how names will be used in advance is all-important, though different in style. Note, too, that I said 'names'. In Locomotive BASIC, you are probably used to working with variable 'names' like Apple, Belong\$ and so on. 'C' also allows you to use realistic names rather than just letters singly or in pairs. This is an advantage in 'C', because using long names does not slow down the action of the program as it does with BASIC. You must, however, define what type of item each name will be used for. This allows the compiler to prepare for each variable that will be used by making the correct allocation of memory. You cannot write statements which in BASIC look like:

```
NAME="SINCLAIR"
```

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

Program structure.

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

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

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

This could **never** be mistaken for a 'C' line. For one thing, 'C' lines aren't normally numbered, although HiSoft C for the Amstrad uses line numbering for your convenience in editing. If you write the lines in the correct order, the order that the compiler will deal with them, there's no need for line numbers. The second point is that the subroutine starts somewhere later in the program, at line 5000. We can't have such a thing in 'C', because the compiler can't use line numbers. Instead of using a subroutine which is called by its line number, 'C' uses a **function** which is called by using its **name**. Users of the BBC Micro (and the QL) are familiar with this idea, and the principle is developed further in 'C'. Even the instruction PRINT is not used in 'C', and the semicolon does not mean 'don't take a new line'. Did I really say that knowing one computer language would prepare you for another?

The point that is really important here, though, is **order**. In BASIC, you can write a core program which has calls to subroutines. It won't run correctly until the subroutines have been entered, but you can place the subroutines anywhere you like in the program. By contrast, defined functions in Locomotive BASIC must be placed ahead of the point at which they are called. If you have a function which, for example, multiplies a price by 0.15 so as to work out the amount of VAT, you must, in BASIC, define it before you use it. You would have lines in BASIC such as:

```
10 DEF FNVat(S)=S*.15

20 INPUT "BASE PRICE";X

30 PRINT"VAT IS ";FNVat(X)
```

because the BASIC interpreter can't look ahead for FNVat. The similarity here is that a defined function in Locomotive BASIC is called into action by using its **name**, FNVat in this example, rather than by using a line number. In a 'C' program all functions must be defined, but this can be done **following** the main program. Like a well-structured BASIC program then, which might consist of a core program of perhaps ten to a hundred lines of main program followed by subroutines, a 'C' program is written in the order of main, then functions. This means that details, such as declaring variable types, all come at the start of the program, followed by the main program, and the functions often come last of all. This is a very logical arrangement as far as the programmer is concerned. It certainly makes the writing of programs much simpler than is the case in other languages. Any modern version of 'C' can be expected to possess a good editing system, so it's easy to add statements at the start or at the end of a program if you have left something out. If you have programmed for a long time in BASIC, you might feel rather lost without line numbers, and HiSoft C therefore allows you the use of line numbers. This is shown only in the editing example in the manual, but you can use line numbers just as you do in BASIC, and the line numbering can be made automatic in the same way as used in Locomotive BASIC. For example, typing **10,10** (ENTER) will cause the line numbering to start at 10 and to change in steps of 10.

The line numbering is not essential in HiSoft C. With the compiler switched in, you can enter a set of statements with no line numbers, and you can then compile and run the program. You cannot, however, record the program to use again. The line numbers are also very useful if you want to list selectively, or delete some lines. You can use a line-editing action which is similar to that of the Amstrad machine. The line numbers, however, are **not** recorded along with the program, and you can find that your program has an entirely different set of line numbers when you replay it. In addition, the line numbers are not printed when you take hard-copy. The examples in this book have been printed directly from the text of working programs, with no line numbering. You can, if you like, prepare your programs using a word-processor, leaving out line numbers, and recording on tape or disc. If you program in BASIC in this way, you will know what is involved. You can also read recorded 'C' programs with a word-processor, or by using the CP/M TYPE command.

The use of line numbering just like BASIC makes it much easier for you to have second thoughts. It's easy, for example, to insert new lines just by giving them suitable numbers. You can, for example, put in lines with numbers 11, 13, 14, 16 and so on, between line 10 and line 20. If you find that you are running out of numbers, then HiSoft C comes to your rescue with the facilities that you normally have on the Amstrad machines. You can, for example, renumber lines with a **higher increment number**. If you find that you are writing so many extra lines that you will need to use many more lines, then you can renumber all the existing lines of the program by typing simply **n** then (ENTER). This will renumber so that the first line number is 10, and the subsequent lines are numbered in tens. You can then continue to insert lines with intermediate numbers, and renumber as often as you like. If you have programmed in BASIC using the popular plan of reserving certain ranges of line numbers for certain tasks (like 10 – 100 for the main program, and each subroutine starting at 1000,1200,1400 and so on), then forget it! This is never needed in 'C', and in any case, a program is always automatically renumbered when it is replayed from tape or disc. You can also delete lines in a similar way. Typing **d100,200** will delete all lines between, and including, 100 and 200. A command such as **d10** will delete a single line. Unlike BASIC, typing a line number and then pressing (ENTER) will *not* delete a line. It will only bring up the polite message 'Pardon?', which is how the HiSoft Editor for 'C' deals with anything it can't recognise. The command **d1,32767** will delete all the lines of a program.

In addition to these useful editing commands which are a normal part of the BASIC of the Amstrad, HiSoft C allows you a couple of commands which are more commonly used in word-processing programs. One is **f**, which is used to find any string, such as a name. You have to specify the range of line numbers over which you want to search. The command **f10,100,copy** (ENTER) will list on your screen (or printer) each line which contains this word **copy**. The line appears on the screen, ready to edit using the arrow and (DEL) keys in the usual way. The range of line numbers that is needed has to be entered only once if you are using the same range for several searches. After the first search, you can use **f,,newone** for example, omitting the line numbers. The other action of this type is the change action. This is used to change one string into another. For example, **f10,100,copy,turn** (ENTER) will list all the lines that contain **copy** and allow you the choice of altering each **copy** to **turn** if you want to. If you want to make the change, you press **CTRL s**, and you'll see the alteration carried out, then the next word will be found.

Source-code and Machine-code.

HiSoft C is sold in two versions, cassette and disc, which operate in almost identical ways. Normally, you will work with the compiler program held in the memory of the Amstrad for all the time that you need to use it. When you type a program, that program exists only as a set of ASCII codes, just like the output from a word-processor. This is called the 'source-code', and is recorded on cassette or disc by using the **p** command, more of which will come later. When a program like this is loaded from the disc or cassette, it is still just a set of ASCII codes. To make the program run, it first has to be compiled, and then set into action.

This can take some time, particularly if the program is a long one. The **reversi.c** which is provided with HiSoft C is a good example of another problem that can arise – it's too long to compile in this way on the CPC464 machine! Another disadvantage, however, is that only an Amstrad which has HiSoft C loaded into it can load, compile and run a program which has been created this way. This is because normal compiling action does **not** record or replay machine code. All that is recorded is a serial file of strings, the instructions of the 'C' program. These have to be read and compiled before running is possible.

There is an alternative however. By making the first instruction of a long program (or any other program you want to use) **#translate filename**, using your own filename, you can compile and record machine code. If the program is a very long one, you record the source-code, including the **#translate** instruction. You then compile the recorded program, using the instruction **#include source_filename**. This loads the compiler into the memory only when it is needed. The program can be read from disc or cassette, compiled, and the resulting machine code **stored back on the disc or cassette**. This allows you to run a 'C' program without having HiSoft C in your machine, and in this way **much longer** 'C' programs can be compiled. Once this has been done, the code can be read from the disc by any Amstrad machine which has enough memory, and run. In this way, a program which you have written in 'C' can be used by **any** Amstrad, even one which is not supplied with HiSoft C on cassette or on disc.

Throughout this book, the examples will be useable in any form you please. As with any compiled language, the main advantages of using 'C' are available only to disc users, and I believe that most programmers in 'C' will be using a disc system, as I am. Nevertheless, if you are learning about HiSoft C with only a cassette system, you will be able to use all of the examples which are included here, though progress will be slow. The programs that are illustrated all use lower-case letters for words of command. You must use lower-case letters for program instructions and for commands to the compiler, because upper-case letters will not be accepted. The use of upper-case letters gives you a set of 'undefined symbol' error messages. Because of this, all references to program instructions in the text will be in bold type. The **w** command will list a 'C' program on your printer if you have a printer connected, and the listings that are reproduced in this book have all been made by using the **w** command to an EPSON RX-80 printer.

HiSoft C summarised.

Unless you have used other varieties of 'C', the advantages of HiSoft C are not necessarily clear to you, even after reading the manual. You can also find that, unless you are a fairly experienced 'C' programmer, even 'C' programs printed in other books may not run when you try them, because you need to know how the Amstrad version works first. The main point is that HiSoft C is a version which corresponds very closely to international standards. You should never have to re-learn your 'C', because there are not the 57 varieties of 'C' that you find with BASIC. If you change your computer for a later design, then you will find that you can use 'C' on the new machine as readily as you did on the old. This is particularly true if HiSoft C is also available for your new machine.

Having praised the advantages of standardisation, however, there are also some non-standard advantages. The Amstrad is a machine which has quite superb capabilities for graphics and sound, better than those of many other machines. HiSoft C comes with a library of functions to allow you to control these features of the Amstrads, and obviously these features would not necessarily be available on other computers. There are also provisions for writing in assembler language, and for direct commands.

Command	Effect
a	Set drive to A.
b	Set drive to B. Will cause a lockup if no 'B' drive exists.
cpm	Switch to CPM, clearing out 'C'.
dir	Display disc directory.
dir file	Display filename, which can use wild card.
disc	Switch to disc in, disc out.
disc.in	Switch to disc in.
disc.out	Switch to disc out.
drive letter	Switch to drive of specified letter, if present.
era name	Erase file of specified name. A wildcard can be used.
ren old new	Rename old filename as new.
tape	Switch to tape in, tape out.
tape.in	Switch to tape in.
tape.out	Switch to tape out.
user number	Change user number.

Owners of the CPC464 should note that these bar commands use the neater style (no '@' strings needed) of command as used by CP/M and also used in the 664 and 6128 machines.

The bar commands which can be used direct from the keyboard during editing.

FIGURE 1.1

The bar(|) commands that are listed in Figure 1.1 can also be used **directly** from the keyboard. | **dir**, for example, is very useful to find what is on your disc at any time before you start saving a program. You also have access to all the rename and erase commands for disc operation without having to go back to BASIC. You can also set the programmable keys for some of the most-used 'C' instructions, saving wear on your typing finger(s).

The use of line numbers when entering 'C' statements makes editing particularly easy, especially when you can make full use of the familiar editing system of your Amstrad. This makes learning HiSoft C very much easier, because you do not have to struggle with an unfamiliar editing system at a time when you are likely to make a large number of mistakes. For disc users who choose to use the Disc version, there are many advantages in HiSoft C. Of these, the main advantage is that the library is much easier to use. This is another feature of 'C' which is difficult to grasp if you have only programmed in BASIC. Every version of 'C' comes with the small set of 'C' instructions built in, but also with a 'library'. The library is a set of recorded functions. It's rather like getting a version of BASIC which came with a disc full of ready-made subroutines. These library functions contain a lot of extensions to the language, not least of all the special instructions that are needed for sound and graphics on the Amstrad machines. When you have to use cassettes, getting these library functions off the cassette and into memory takes rather more time than most of us can spare. When the library is held on disc, the need to wait is much reduced. In addition, when you make use of discs, the full range of disc filing commands can be used in programs.

Chapter 2

Starting 'C' programs.

Loading in.

The 'C' disc or cassette is loaded into your Amstrad in much the same way as any other disc which contains machine code. This means that the 'C' compiler is recorded in the form of an unprotected BIN file, which is loaded and run by using `RUN"hisoft-c"` (ENTER). Once this has been done and the compiler is in action, your only way out is back to BASIC, by pressing `ESC SHIFT CTRL` together. This will re-start BASIC, and remove all of the compiler code. The disc contains several other sections of code, including the library routines, which are arranged to load in as and when necessary. These other sections are held as 'C' source files, which means that they are in ASCII code, and can be read by the `CP/M TYPE` command or the `g,filename` command of the 'C' editor. To see these files using `CP/M`, and to print them out if you have a printer, start by switching back to BASIC. Insert the `CP/M` master disc, and then type `| CPM` (ENTER) to get into `CP/M`. Now remove the `CP/M` disc and insert your HiSoft C disc. To see the file `extcmd.h`, for example, type `type extcmd.h` and press (ENTER). You will hear the disc spin, and see the file appear on the screen. If you have a printer connected, then using `CTRL P` before entering the `TYPE` command will cause the file to appear on paper. The alternative is to start 'C' running, use the command `d1,32767` to make sure that the memory is clear, then use `g,filename` to get the file. The file can be printed out using the `w` command. You can then take a look at a professional class of 'C' program. Don't let it put you off!

You have paid for the master copy of 'C' on cassette or on disc, and there is no reason why you should not make backup copies of the disc. The HiSoft manual shows how to make copies of the library routines from the cassette, and making disc copies of the main program and the library routines is just like copying the `CP/M` master disc. Certainly if you intend to be using 'C' for many years, it would be preferable to have *at least one backup* for the disc. Your backup should contain all of the files except `reversi.c`, and you can leave the backup disc unprotected, so that you can add your own 'C' files to the list. It's a good idea to have the HiSoft C and the library files on each disc that you use, to save having to use the master copy. If you use cassettes, then you have to swap the cassettes around a lot in any case. Remember that once you have entered 'C', you cannot use familiar commands like `cat` or `list`. You can use the bar (`|`) commands in the usual way, however, with the usual reminder that if you use `| CPM`, the `CP/M` master disc must be in the drive, and you will then lose the 'C' compiler. You can, however, use commands like `| ren` and `| era` in their `CP/M` form.

First steps.

All 'C' programs can be constructed in the same way, and though you can leave out some steps in HiSoft C, it's advisable to keep to the rules of standard 'C'. If you do, it's much easier to write 'C' programs for practically any machine. In addition, it helps a lot if you write programs the way that you ought to design them, outline first and details later. Remember also that you need to keep to Amstrad rules as well. For example, you must, if you are not using the automatic line numbering, remember to leave a space following a line number. Failing to do this will make your 'C' programs look **very** peculiar. The simplest possible outline of a 'C' program then, is:

```
10 main()  
20 {  
30 /*statements;*/  
40 }.
```

– and we now have to take a look at this to decide what is important in these few lines. One thing which is **very** important is the way that the semicolon is used. In BASIC, the semicolon is used to ensure that printing is to be kept on the same line. In 'C', the semicolon is used as a separator, showing the end of a statement. This is the sort of thing you do in BASIC just by ending a line and taking a new line number. In 'C' you can, for example, use the semicolon to separate statements in the same line, as you use the colon in BASIC. Omitting the semicolon is a way of instructing the compiler that there is more of a statement to come. At this stage, it's a bit pointless to describe the rules, because until you have had some experience in writing programs, you won't really see why semicolons are used in some places and not in others. For that reason, I'll point out in each of the early examples the few instances in which a semicolon has not been used where you might expect it.

In the example of program outline, the first line consists of the special name **main()**. A 'C' program consists of a set of named **functions**, using whatever names you like to give them, but there must always be one that is called **main**. The brackets are an essential part of this, and it's not very often that you need to put anything between these brackets. For other names of functions, though, you will want to place various items between the brackets. In any case, you can't omit them. The **main** program is the one that calls up all of the other functions, just as a BASIC core program can call up various subroutines. The curly brackets then show the start and the end of the **main** program. The { indicates the start, and the } shows the end. You don't need any other way of marking the end of the program.

The simplest possible program is then written using keywords. A keyword in 'C' is rather like a keyword in BASIC, it is reserved for a special purpose and you can't use it for anything else. Keywords must be correctly spelled, otherwise an 'Undefined symbol' error will be announced **after** you have compiled and when you try to run the program. This means that any of the errors which you would think of as 'syntax errors' in BASIC are very often not discovered in a 'C' program until after compiling. This wastes a lot of time, so you need to be rather careful about checking what you type. It isn't made easier by the difficulty in reading the Mode 2 lower-case printing on the Amstrad screen, either. Figure 2.1 shows the keywords of HiSoft C.

auto	Specifies type of variable or function.
break	Break out of loop.
case	Marks a choice made by using switch .
cast	Change the type of a variable.
char	Character variable type.
continue	Go to start of loop.
default	Select default option in switch .
do	Start of do..while loop.
double	Double precision variable, not implemented in V1.2.
else	Alternative in if statement.
entry	Not implemented.
extern	Used in V1.2 to declare non-integer functions in advance.
float	Floating-point variable, not implemented in V1.2.
for	Start of counter controlled loop.
goto	Jump to position of label name.
if	Test word, used along with else .
inline	Used in V1.2 to head list of machine code bytes.
int	Integer variable type, range -32768 to +32767.
long	Double size variable type, not implemented in V1.2.
register	Variable type, not implemented in V1.2.
return	Pass back value of parameter from function.
short	Normal variable type, only type used in V1.2.
sizeof	Measuring number of bytes in variable.
static	Type of variable using fixed memory.

struct	Compound variable consisting of several fields.
switch	Passes control to one of a number of statements.
typedef	Defines a name as meaning a variable type.
union	Variable type which can be one of a defined group.
unsigned	Number in range 0 to 65536.
while	Marks start of while loop, or end of do loop.

The keywords of HiSoft C. Some, such as long and float, are not implemented in the first version.

FIGURE 2.1

Note that most of these, apart from **cast** and **inline** will be found on other versions of 'C', but other versions, notably the 'C' for the IBM PC, have a few more keywords, in particular for random access disc filing.

The word **main**, note, is *not* one of the keywords. This doesn't mean that you can use it for anything you like, but it is a title for the main program, not a word which describes an action. This is an example of a word which is an **identifier**. In BASIC, the only identifiers that you use are filenames and names for variables. 'C' uses a lot more types of identifiers, and they are used in much more interesting ways. In this case, the word 'main' identifies the main program, and you could use other words to identify the functions (the 'C' replacement for subroutines) which are called up by the main program. You also use identifiers for other things, like variable names, subject to a few rules. The rules are that an identifier must start with a letter, with upper-case and lower-case being treated as identical. Since you must use lower-case for keywords, it makes sense to stick with lower-case for identifiers too. Professional programmers use upper-case letters in identifiers which are present for special purposes, as we'll see later. You can then follow this letter with other letters or with digits, but no blanks or punctuation marks. The only character that is allowed, apart from letters and digits, is the underscore (), which you get by pressing the SHIFT 0 key. The underscore is useful as a way of making long names more readable (like name of item). You could, if you wanted to, start a word with an underscore, but there again its better not to, because this could lead to trouble later on in your 'C' programming career. The reason is that words which begin with an underscore are used within the compiler, and unless you can be sure that you are using different words, you can cause problems. You **can** use names of more than eight characters, but **only the first eight characters will count**. This gives you rather less choice about things like variable names than you have in BASIC, because Locomotive BASIC places no restriction on name lengths up to 40 characters long.

You probably know that in Amstrad BASIC, there are some variable names that you cannot use, such as PRINT, TAB and any other reserved word in upper-case letters. HiSoft C, like other versions, has some identifier names which are already allocated.

blt	Move bytes in memory.
define	# command.
direct	# command.
error	# command.
fclose	Close file.
fopen	Open file.
fprintf	Print to file.
fscanf	Read from file.
getc	Character from file.
getchar	Character from keyboard.
include	# command.
isalpha	Test for letter.
isdigit	Test for digit.
islower	Test for lower case.
isspace	Test for space.
isupper	Test for upper case.
keyhit	Test for key struck.
list	# command.
main	Marks main program.
printf	Print on screen.
putc	Send character to file.
putchar	Place character on screen.
rawin	Read keyboard for key.
rawout	Send code to screen.

scanf	Read variable value from keyboard.
sprintf	Send string to file.
sscanf	Read from string into other variables.
swap	Exchange variable values.
tolower	Alter character to lower case.
toupper	Alter character to upper case.
ungetc	Put character back on file.

Identifiers that are already allocated. These belong to functions which are built-in to the HiSoft compiler.

FIGURE 2.2

These are listed in Figure 2.2, and when you look at this list, you might think that these were another set of reserved words, as many of them would be in BASIC. The difference is important. All of these identifier names **can** be used by you for something else if that's what you want. If, for example, you want to call your program **gets** or **puts** then you can do so. You would be foolish to do this, because by changing the meaning of these names, you are losing the use of some action that you might need, but you will not cause any error message. The difference is important, because if you try to use a reserved word for anything else, the error will be signalled; if you use one of the 'predefined' identifier words, there's no error, and you won't be informed. You may wonder, however, why some action later turns out to be impossible! The words in Figure 2.2 are the names of the library routines of HiSoft C, and each of them will call up a routine which may be part of the compiler (an internal function), or from the library on disc or cassette. This allows you to incorporate ready-written pieces of 'C' into your own programs, saving a lot of reinventing the wheel.

Following the **main()** identifier, there is a newline (obtained by pressing the (RETURN) or (ENTER) key) and an opening curly bracket. This must be present, and if you omit it you will see an error message, usually 'Bad declaration', when you try to compile. The opening curly bracket marks the beginning of any program or piece of program, and follows the program name and declaration of variables. This is something that we'll look at very shortly. The next line is where we would expect the program to do something. Instead, all that we have is /*Statements*/. The combination of the forward slash (under the question-mark) and the asterisk (with no space between them) has the same effect as REM in BASIC. It marks a piece of the program which is just a reminder to the programmer. Unlike BASIC, you have to mark both the beginning **and** the end of the remark. In addition, a reminder in 'C' does not slow down the program in the same way as a REM in BASIC does. This is because the compiler ignores the reminder and no code is generated. In BASIC, a REM still has to be looked at each time the program runs, just to read the code that means REM. In this line, I have put a semicolon to remind you that there will be a semicolon following each statement. Finally, the 'C' program ends with the closing curly bracket. The pairs of curly

brackets can be in many places in a 'C' program. This is because each section of a program has a beginning and an end, and the curly brackets are used to mark them. Any but the simplest 'C' program will be written as a set of named functions that will be called by the main program, and each of these procedures will have an opening curly bracket and an ending curly bracket. If, incidentally, you miss out the ending curly bracket in the **main** program, you may find that the error is not picked up by the compiler. The program will compile, but it will not run. In the example, it will stop with the error message 'expecting a primary here'. The main reason for this is that it can't think of any other name for the error!

You can now check the sample program. If you have started by typing **i10,10** for automatic line numbering, you will have to leave this by pressing (ESC) after the last line of the program. If you need to edit a line, then type **e** followed by the line number and press (ENTER). Use the right shift and (DEL) key to repair the damage, and press the (ENTER) key again when you have finished. Use **l** (letter ell) to get a listing of the program to check it. Once you are sure that it is perfect, you can compile it. This is done by typing **c** (ENTER). This clears the screen, setting 80-character lines again if you had switched to 40 characters for readability, and just waits. Using **c** just switches the compiler in, it doesn't start it. To start compiling, type **#include**, and (ENTER). If you have made no mistakes, you will see the lines of the program appear in order, followed by the cursor. If you do find errors in this example, then it's nearly always going to be omission of a curly bracket or a slashmark. You then need to prepare for running. The compiler pauses at the end of compiling so as to allow you to put in special commands to the compiler, but in most cases this isn't needed. To indicate that you want to use the program, you press (CTRL Z). This brings up the message 'Type y to run', and pressing the 'y' key will run the program. If a mistake like a missing final curly bracket has been found, you'll get an error message in place of the 'Type y to run' message. You can then run the program by typing **y** without needing to press (ENTER). Since the program doesn't do anything, nothing appears on the screen except another invitation to run the program by pressing the 'y' key. Pressing any other key puts you back into the hands of the editor.

Output to the screen.

You have probably already noticed that 'C' does not have a reserved word **print**. There is, in fact, no reserved word for the action of putting something on the screen. This action is one which carries an identifier name for a library routine rather than being one of the reserved name actions. The identifier word that you need is **printf**. Unlike most of the library routines, **printf** is built-in as part of the compiler code, so that the routine does not have to be read from the disc each time you compile it. The use of **printf** is, however, quite different from the use of PRINT in BASIC. The name **printf** has to be followed in brackets with details of what has to be printed. This means not only what you want to print, but also how you want it printed, formatting as it's called. The formatting commands and the items that you want to print are all included within brackets, with quotes around the formatting commands and any characters that are to be printed. What is 'written' on the screen in this way can be a number or it can be text. The simplest possible examples of text writing look sufficiently like BASIC to be easily understood when you are reading a 'C' program. Figure 2.3 shows an example, which you can type, compile and run.

```

main()
{
printf("\nWords");
printf("\n%d", 5+3*2);
}

```

A short program in 'C' to get you familiar with compiling.

FIGURE 2.3

In this example, the actions are of writing a word and performing a piece of arithmetic. The writing of a word is rather different to the PRINT "Words" that would be used in BASIC. The word has to be placed between quotes, and also has to be placed between brackets. The semicolon at the end of the line has nothing to do with the printing action, remember, it's just the signal to the compiler that there is more to come. The real novelty here is the `\ n` which appears within the quotes and just ahead of **Words**. The `\` is the backslash sign, which is on the key next to the right-hand SHIFT key. Don't mix this up with the forward slash next to it which is used in `/*rem*/` lines. The effect of `\ n` coming before the text is to force a newline before anything is printed. You could also place another `\ n` after the text to cause a new line to be taken after printing. There is a complete set of these backslash instructions, all of which must be included between quotes in a **printf** type of statement. Figure 2.4 shows this set.

Mark	Meaning
<code>\ n</code>	Newline (ENTER/RETURN key).
<code>\ t</code>	Tab (one space default).
<code>\ b</code>	Backspace.
<code>\ r</code>	Carriage return (not newline).
<code>\ f</code>	Printer formfeed, screen clear.
<code>\ ' </code>	Put in single quote.
<code>\ " </code>	Put in double quote.

Any other codes can be put in as numbers **in octal code** following the backslash. For octal codes, see Appendix A.

The specifier letters which can follow the backslash.

FIGURE 2.4

If, for example, you use `\ f` this will carry out a formfeed if sent to the printer, and will clear the screen if the screen is being used. In the next line, what is written is still placed between

brackets, but there are no quotes. The arithmetic **result** is printed out, just as it would be by `PRINT 5+3*2` in BASIC. As in BASIC, the multiplication is carried out before the addition, so that the result is 11, not 16. Here again a semicolon has been used to mark the end of the statement, and there's another `\n` used to cause a newline. The novelty this time is the `%d` which follows the `\n`. The `%` sign is a general way of indicating how you would like a number printed, and when it's followed by a `d`, then the number is printed in denary. If you haven't come across this term before, it means the ordinary scale-of-ten numbers that we use. Once again, there is a whole set of these 'number specifiers', and Figure 2.5 shows the complete list. After this line, the main program ends with the curly bracket.

Mark	Meaning
<code>%d</code>	Signed denary number, range -32768 to +32767.
<code>%u</code>	Unsigned denary number, range 0 to 65535.
<code>%o</code>	Unsigned octal number.
<code>%x</code>	Unsigned hexadecimal number.
<code>%c</code>	Single character.
<code>%s</code>	String ending with a zero.
<code>%%</code>	Print <code>%</code> sign.

Quantities are normally printed right-justified, but using a negative sign before the specifier letter will force left justification. Each specifier letter can be preceded by a number to set minimum field size, 0 will print a leading zero or blank.

The formatting code letters which can follow the `%` sign.

FIGURE 2.5

This very simple program nevertheless illustrates a lot of important points about 'C'. The most important point is that the program consists entirely of calls to functions. There is absolutely no processing in the main program, simply two calls to the `printf` function. The brackets, which we did not use in `main()` are used in `printf` to carry the items that we want to print, and also the instruction codes about how we want it all printed. This is the way of carrying out most actions in 'C', and very often we have to write our own functions if there is nothing suitable in the library.

Now press the 'c' key to get to the compiler, type `#include` to compile, CTRL z to signal ready to run, and answer 'y' to the 'Type y to run:' message. You must, of course, press the (RETURN) (CPC6128) or (ENTER) (CPC464,664) key each time. When your program has been compiled and will run, it's a good idea to check recording. To record your program on disc or tape, type `p10,50,test`, then (ENTER). You can, of course, use your own filename in place of `test` here. When the program has been saved, you can load it with

g,,test , specifying the filename again. Note that the two commas **must** be used. If you have only one drive, you don't need to specify drive number with g, but you must use the correct file name. You will find that the program is always renumbered when you list it, with numbers starting at 10. This is not obvious in this example, but if you add some comment lines such as:

```
25 /*odd line number 25*/
```

and then record this and load it in again, you will find that the renumbering has been carried out.

If you use a word-processor program for writing and editing your 'C' program files, you will not make use of line numbers until the program is loaded for compiling. You will see from the disc directory that your 'C' program has been recorded as a file which is indistinguishable from any other file in ASCII codes. This file is called the 'source-code'. Until this source code has been compiled it is just a file of ASCII codes, nothing more. Once compiled, it is object code, closer to machine code and quite different in action. The most important difference from your point of view is that the source code can be read, edited, and is easily understood. The object code has no meaning unless you know about machine code, is very difficult to edit, and can be recorded only when you are using the **#translate** instruction inside the program code. The routine for **#translate** compiling is summarised in Figure 2.6.

1. Write the source code and test it thoroughly.
2. Edit in a first line which is **#translate filename**, using the filename that you want for your machine code. Make sure that this name does not appear on any other file on the disc.
3. Save the source-code on a disc.
4. Use the **#include** sourcefile command to read the source code from the disc and compile it. The code will be compiled to machine-code, and stored on the same disc under the filename that you used along with **#translate**.

A summary of the procedures for using **#translate** to make a 'standalone' program.

FIGURE 2.6

Even if you use a cassette system, there is no point in using **#translate** for short programs. This is because even a short program requires a large amount of code. The example of Figure 2.3 requires about 3K of code when it is compiled to a combined BASIC and machine code program which will run independently. Once the program has run, the machine instantly returns to BASIC, and unless your program ends with a loop of some kind, the results of running it are not visible. The source code for this sample program fills only part of one sector, 1K on the disc. The reason for the difference is that when you

compile and run in one operation, most of the code is already in memory, either in the RAM or read from the disc or cassette. When you use the **#translate** instruction to create a program which will run on **any** Amstrad, it has to include all of the code (the 'run-time system') that would normally be held in memory as part of the 'C' compiler program. The consolation is that longer programs do not necessarily need very much more code! From now on, we will concentrate on examples which use ordinary compilation.

Using constants.

When you use a constant, like 3.1416, in BASIC, you are always advised to assign the value to a variable name. The reason is that this avoids the BASIC interpreter having to convert the ASCII codes for the number into number-variable form each time the number is used. The same is true of 'C' programs, but with the difference that you can either assign to a variable name or use a **#define**, of which more later. As far as HiSoft C is concerned, one of the things that you have to get used to is that you can't use numbers like 3.1416 – not with V1.2 at least. The reason is that HiSoft C V1.2 does not allow what are called 'floats', numbers which can contain fractions. This is one of the few missing parts of a complete 'C', and we simply have to accept it. Originally, 'C' was designed to be used by programmers who were writing compilers for other languages and for operating systems, and only integer (whole) numbers were needed. Though standard 'C' can cope with fractions, HiSoft C can't in V1.2, though there are hints in the manual that this extension will follow.

Even with this omission, 'C' has rather more different types of data than BASIC, and one of these is the integer, which uses the reserved name **int**. As we saw briefly in Chapter 1, you can declare that a name will be used for an integer, and then assign a value to the integer. The syntax of declaration is:

```
int penta;
```

using the reserved word **int** followed by the identifier name **penta**. You can have several such declarations on the same line, with commas following each name. For example, you could have a line:

```
int penta,hex,hept;
```

if you wanted to declare several names as integers. Note the semicolon to show the end of the statement, the end of that declaration. Once the names are declared, you can make assignments to these names, using integer numbers.

```
main()
{
int hex;
hex=6;
printf("\n%d times 2 is %d",hex,hex*2)
;
}
```

Using a variable, in this example an integer variable called **hex**.

FIGURE 2.7

Figure 2.7 shows a simple program which makes use of the integer **hex** to mean 6. In this example, the declaration of the integer and its assignment are both straightforward, but the **printf** line is not. In 'C', printing is a very different kind of operation as compared to BASIC. The first part of the **printf** statement consists of the words and formatting instruction only. We want two numbers to be printed, both in denary form. In the phrase that is to be used, then, the **%d** is put in each part where a number will be printed in the version we see on the screen. Once the quotes are closed, the numbers are put in, using the same left to right order, and with commas used to separate the numbers. The numbers are **hex**, the integer, and **hex*2**, the result of a calculation. Once again, this is a statement, and it has to end with a semicolon. When it prints on the screen, you see the message:

```
6 times 2 is 12
```

which is not exactly world-shaking, but until you get used to the way in which 'C' uses its **printf** statement, it's an example you'll probably need to consult now and again.

Constants.

The use of a variable for holding a number in 'C' is close enough to the methods of BASIC (so far) to cause you little worry. There is an alternative in 'C', however, for storing items which you might want to use in any part of a program. These items are called, logically enough, constants, and they have to be defined in a way that is quite different from our definition of variables. The definition of a constant is done at the beginning of a program, before the **main()** portion or (almost) anything else. The syntax is simple enough, **#define**, followed by a space and then the name that you want to use, another space and the value. *There must be no semicolon at the end of a #define line.* Constants can be numbers, single characters or strings, as you please, providing you assign correctly and use correctly. In other versions of 'C', this part of the programming is handled by a separate section, called the 'pre-processor'. In HiSoft C, it's all part of the main compiler action, and we'll treat it as such. Take a look, for example, at Figure 2.8.

```
#define foot 305
main()
{
  int ft;
  ft=3;
  printf("\n%d feet is %d millimetres",ft,
  ft*foot);
}
```

Using a constant, declared with **#define**. This is one of the 'pre-processor' actions.

FIGURE 2.8

The line which 'declares the constant' of **foot** is situated immediately at the start of the program, using **#define foot 305**. In the main program, the part which lies between the curly brackets, we will use **foot** as meaning the number 305, the approximate number of millimetres in a foot. This meaning **must** be declared before the program begins. This way, the compiler has allocated memory space for the constant and is ready to use it before the program needs it. As we have seen, you might have to allocate several constants like this before a program starts. These constants need not all be integer numbers like 305. They could be letters or phrases, like 'Press any key', and this use of a constant replaces a lot of the purposes for which we use string variables in BASIC. This is important, because 'C' does not have string variables **in the form that we use in BASIC**. In the example, you can see the **printf** phrase "`\n%d feet is %d millimetres`" used with the **%d** to specify where the numbers will be printed, as denary numbers. Following the phrase comes the quantities, the integer variable **ft** and the constant **foot**. In this example, the numbers have been printed as denary numbers, but you can **force** any **printf** action to produce numbers in other forms, such as hex or octal, that you want. You can also decide how much space you want the number to take up. Try a change to line 60, so that it reads:

```
60 printf("\n%-6d feet is %8d millimetres",ft,ft*foot);
```

and compile and run this one. You will see that the figure '3' appears on the left hand side of the screen, and the number 915 is spaced out from the word 'is', taking up 8 character positions. As you may have guessed, the figure 8 along with the 'd' specifies that the number shall be printed taking up 8 character positions, and placed at the right hand side (right-justified). By using a minus sign in front of the number, the space is allocated similarly, but the number is set over to the left (left-justified). This type of control over number position is called 'fielding', and it's much easier in 'C' than it is in BASIC. If you don't use any numbers along with the 'd' in formatting, a number will simply take up whatever space it needs in the **printf** statement. You can, of course, decide on the number of spaces either ahead of or following the number by putting them in with the spacebar. The fielding method is particularly useful if you want numbers organised in columns, and really comes into its own when fractions can be used. In a later version, perhaps...

I said that a constant did not have to be a number, but could be a character or a string. For items like that, you still use **#define**, with the name that you want to allocate, and the character or string spaced from it.

```
#define mesg "press any key"
#define key "Y"
main()
{
printf("\nuse the %s key or",key);
printf("\n%s", mesg);
}
```

Using string and character constants with **#define**.

FIGURE 2.9

The character or string needs to be surrounded by quotes, as Figure 2.9 shows. If you omit the quotes you will get an error message during compiling when the character or string is used, rather than where it is defined. The message will be ‘undefined variable’, but if you surround the characters with marks other than quotes you can get some quite exotic error messages. The other part of the deal is that if you want to print messages in this way, the **printf** statement needs to be changed. In place of the **%d** that you needed to specify a number in denary form, you need to use **%s** for a string or **%c** for a single character. Without these modifiers, **nothing** gets printed! In the example, **%s** has been used for both, but we could have used **%c** for the single character. The use of **#define** in this way allows a lot more than just the occasional number constant or message phrase. With **#define**, you can make your programs much more readable, particularly by definitions such as **#define white 0** and **#define black 1**, which allow you to use words in place of numbers for items such as board games, or allocate values to items, as in **#define mayfair “£5000”**.

More variables.

We have already made use of an integer variable in a program, and the style is easy enough. However, there’s much more to this type of variable than meets the eye. A variable which is declared as, for example, **int num** is what is called an ‘automatic’ variable. In all varieties of ‘C’, you can state this by typing **auto int num**, but if you don’t use any word before **int**, then the use of **auto** is assumed (it’s the default). Most of the variables that you are likely to use in HiSoft C programs will be auto types, simply because of convenience. The alternative, as far as HiSoft C is concerned, is a **static** variable. Now the difference is not at all easy to understand if you have only ever programmed in BASIC, and a more extended explanation will follow later when we deal with functions. The difference concerns how values are stored and used. The storage of a static variable in HiSoft C is more efficient and the value can be reached much more quickly. You might, for example, want to use static variables in games programs to get the highest possible speed. As far as programming is concerned, though, the important difference is retaining values. All variables in HiSoft C are local. This means that they have a meaning only within a function in which they have been declared, and in any function contained within that. So far, we have used only a **main()** function, and the point is not important – yet. When a variable is declared in a function that is called by **main**, any value that was allocated to it disappears whenever the function ends. You can use the same variable again in **main()** with a different value without confusing the machine. If you return to the function, the assignment starts all over again. If the variable is **static**, though, its value is held waiting to be used again. If you return to a function in which a variable has been declared as **static**, and in which it had the value 5, then the value of 5 will be assigned to it whenever the variable is used again. There is an illustration of this in Chapter 10. For the moment, however, the **static** option is one that you should think of simply as a convenient way of speeding up a program, and we’ll look at the other implications later. One feature which might be useful right away, however, is initialisation. The way that you can declare and assign in one operation in BASIC, such as **A%=5**, is very useful. In HiSoft C, you can’t do this with automatic variables, but you can with statics. Take a look at the very brief example in Figure 2.10.


```

main()
{
static int dot=6;
printf("\n%d",dot);
}

```

Combining declaration and assignment for a static integer.

FIGURE 2.10

This declares a static int **and** makes the assignment of 6 to its value. The **printf** statement then shows that the assignment has been carried out. If you attempt to do this with an auto variable, as by deleting the word **static**, then you get a **RESTRICTION** error message. As this suggests, this restriction is peculiar to HiSoft C, and you can initialise automatic variables in some other versions of 'C'. It's really another good reason for preferring static variables. If you have read some other books on 'C' (and the best of luck!), you may have seen references to **register** variables and **extern** variables. These are not available in HiSoft C, and I don't think you'll miss them!

Getting a value.

Suppose that we extend the use of an integer variable to a variable whose value is entered from the keyboard? One of the standard identifier words for reading an input is **getchar()**, which is a function that is built-in. Being built-in means that we don't have to worry about the complications of reading source-code from the library. It's the old Sam Goldwyn motto, simplicate and add lightness, in action! Using this built-in function, however, brings us up against one of the features that newcomers to 'C' find irritating – the use of characters. The function **getchar()** will get characters from the keyboard, meaning that you can type any character, digit or letter that you like. If you look up the action of this function in the HiSoft Manual, however, you see it described as **int getchar()** meaning that it gives you (or **returns**) an integer. Whatever you type at this point is accepted as an ASCII code, and this is the integer value that you get. What we are going to type is a number which will have to be assigned to a variable name of **x**. The value of **x**, however, will be an ASCII code. For the numbers 1 to 9, this means a code in the range 49 to 57. We can get the number values back from this by subtracting 48, the value of ASCII '0'. Now we can do this in two ways.

```

main()
{
int x;
printf("Enter a number, 1 to 9 \n");
x=getchar();
x=x-48;
printf("\nSquare is %d",x*x);
}

```

A program which obtains a character from the keyboard.

FIGURE 2.11

In the printed program of Figure 2.11, it is shown in the familiar BASIC way, as `x=x-48`. We can, however, also write it as `x=x-'0'`, meaning that we subtract the ASCII code for zero. This second form is a lot easier to use and understand – for one thing, you don't have to strain your memory for the ASCII codes!

Using `getchar()`, as you'll gather, is rather primitive. Though you can type more than one digit, the function works only on the first, which is why the program limits input to the range 1 to 9. There is a function called `atoi()` in the library for converting characters into numbers, but that's for later. There is also, in the library, a routine which corresponds more closely to BASIC's INPUT, but without the facility to mix questions and input like INPUT "Answer: ";a\$ in BASIC. The trouble with using these routines right at the start of your conversion to 'C' is that they involve a lot of new ideas, and we can't ever take in too much at one time.

Other variable types.

By the time any book on BASIC has reached this stage, the subject of string variables would have appeared. Now strings have an important part to play in 'C', as they have in any language, but the way that strings are handled is not quite so simple if you are making a transition from BASIC to 'C'. The reason is that STRING is not a pre-defined variable type; it isn't in the list of ready-made identifiers of Figure 2.2. We've looked already at how we can use a string in a `#define` line, and when you think about the way you use strings, this probably takes care of more than 60% of the ways in which you use strings in most of your programs. Later, we'll look at how this type of identifier can be created, but for the moment we'll look at the characters that make up a string.

HiSoft C, in common with all others, has a variable type called `char`, which means any character of the computer which is represented by an ASCII code. In HiSoft C for the Amstrad, this includes the graphics characters as well as the ordinary alphabetical and digit characters. Now with this `char` variable, we can do the actions that you associate with PRINT CHR\$() in BASIC, and some more. Take a look at Figure 2.12, for example.

```
main()
{
static char m1=249;
static char m2=250;
static char m3=251;
rawout(4);rawout(1);
printf("\n %c %c %c", m1, m2, m3);
}
```

Printing three characters on the screen using `rawout`.

FIGURE 2.12

Three static char types are defined, variables m1, m2 and m3. These are assigned with ASCII codes 249 to 251, just to rub in the point that the whole of the Amstrad ASCII codes can be used. The use of **char** means that the variable consists of one ASCII code, stored in one byte of memory. Printing these codes will give the screen patterns that correspond to them, the 'running men' shapes. As it happens, however, these will be in the Mode 2 size, which makes them very difficult to see (like the text). The line which contains **rawout(4);rawout(1);** will switch the screen to Mode 1, making both text and character shapes a lot easier to see on the screen.

1. In edit mode, type **#direct+** and press (RETURN)/(ENTER) key.
2. Type **rawout(4);rawout(1);direct-;** then press (RETURN)/(ENTER).
3. You can now enter your source code in Mode 1, which is much easier to read. The mode will return to Mode 2 when you compile, but this also can be changed, see later.

Switching over to Mode 1 with a **#direct** routine. The mode will return to Mode 2 when you compile.

FIGURE 2.13

Figure 2.13 shows how you can switch over to Mode 1 by a direct command, making it easier to type and check your listings. Note that the screen mode will always change back to Mode 2 when you press 'c' to compile, and a fix for that particular problem is listed in the following chapter.

Finally, what about using **rawout()** for characters. The **rawout** function has to be provided with an ASCII code between its brackets – this is the 'argument' for the function. Whatever this ASCII code represents will then be 'printed' in the usual Amstrad way. As far as the use of codes 4 and 1 are concerned, these are the codes for notifying a mode change and specifying that the mode is Mode 1. We could just as easily use **rawout()** with the variables m1, m2, and m3, *even though these have been defined as characters rather than integers*. This is the kind of flexibility that is the joy of 'C' programmers and the despair of academics! The only snag with using **rawout()** is that you don't have the same control over how the characters are presented on the screen as you have with **printf()**. Try removing the **printf** line, and substituting:

```
rawout(m1);  
rawout(m2);  
rawout(m3);
```

to see the effect. It still prints on the screen, but the characters are closed up against each other.

Chapter 3

Functions and the Library.

Functions.

We have seen already that a program consists of any **#define** constants, then a **main()** which is followed by the opening curly bracket, some statements which end with a semicolon, and then a closing curly bracket. In the main program, between the curly brackets, you will place all the actions, in sequence, that the program will carry out. Now in short programs these actions will be simple ones, and they can all be written between the curly brackets of the main program. As your programs become longer, however, you will need to break them into sections, if only for the sake of planning. Just as you can break a BASIC program into a core and a set of subroutines, or functions, you can break a 'C' program into a main block and a number of functions. The similarity between the languages ends there, though. A function in 'C' is called into action by using a name, its identifier. In addition, values can be passed to a function in ways that are not used by subroutines. The use of a function is therefore something that needs rather more thought than the use of a subroutine. Unlike BASIC, 'C' permits only functions, and there is nothing remotely like a subroutine. If you have programmed with functions in Locomotive BASIC, you will feel very much at home with the way that 'C' uses functions.

Take a look for example at the program in Figure 3.1.

```
main()
{
printf("\nThe name is ");
attention('I');
}
attention(n)
char n;
{
rawout(7);
printf("%c",n);
printf(" Sinclair.");
rawout(7);
}
```

A program which makes use of a new function.

FIGURE 3.1

This starts in the usual way, and prints a phrase, 'The name is '. It then calls a function **attention**. Now this isn't a function that is built in to the compiler, nor held in the library.

It's a function that we have to write for ourselves. The name of the function is **attention**, and it will make use of anything that is enclosed in the brackets. What is in the brackets in this example is the character 'I'. Note that this is 'I' with a single apostrophe, not "I" with quotes. The difference is important. The 'I' is a character with ASCII code of 73, whereas "I" would represent a string – and a string is not a single character as we'll see later. The 'I' is the argument of function **attention**, the value that has to be passed for the function to use.

Now of course, we can't compile this and run it until there is a function **attention** written. The **main()** program is ended by the second curly bracket, and following it we type the name of the function, which is **attention(n)**. We can make the letter inside the bracket anything we like, it can be a complete name or a single letter. We then have to declare that we will be using this quantity **n**, and that it's a character. The curly bracket then opens, the **rawout(7)** delivers a beep (equivalent to `PRINT CHR$(7)` in BASIC, remember), and the character is printed. Now this is represented in the first **printf** line as **n**. This is a variable name which exists only inside this function, and because **n** was used in the 'header', as the argument of **attention()**, it takes the value that was used as an argument when the function was called, which is 'I'. The **printf** modifier is **%c**, meaning that the variable which follows will be printed as a single character. The next line is a conventional **printf** – notice that we *don't* need a modifier because we don't want a new line, and we aren't printing a variable, just a phrase within quotes. The last part of the function is another beep, which in fact will just sound as part of the first one because there is so little time between them.

Now when you compile and run, you see the complete phrase appear on the screen. It would, of course, have been just as easy to use the whole phrase in the **printf** line of the main program, but it's always easier to see how something works from a simple example than it is from a difficult one. The important point is that any function you want to use can be called up by using its name, and the brackets are used to pass any arguments to the function. These arguments can be integers, characters or strings, direct or as variable names. When the function itself is written, you write it like another **main()** type of program. You need to declare any variables that you want to use inside the function, and that includes the name that you have used for the argument. You then carry out whatever actions are needed. In this example, this has meant calling up other functions, **rawout** and **printf**, which exist in the memory of the machine along with the rest of the compiler. This is typical of the way that we write programs in 'C' – a small **main()** program calls up functions, which in turn call other functions and so on. Unless you use cassettes, you can use the CP/M TYPE instruction to list the version of Reversi that you get with the HiSoft compiler, and you'll see from that how remarkably short the main program is.

The variable **n** exists only within the **attention** function. If you try to print its value near the end of the program, following the **attention('I');** line, you will find that you get an error message about 'undefined variable' when you try to compile. This means that **n** was not declared as a variable at the start of the **main()** program. The fact that it was declared as a char type in the function refers to the function only, and you can print the value of **n** at any time while the function **attention** is running. The significance of this is that you provided a value of 'I' as the argument for **attention**. This value is then **temporarily** transferred to variable **n** for the duration of the function only. This is completely automatic, and is rather different from the methods that you have to use in older varieties of

BASIC. The name that is used for **n** and anything of this kind is a *local* variable, and there is no equivalent to the use of this type of variable in Locomotive BASIC. All variables that are declared inside a 'C' function definition are automatically made local. We'll look later at the principle of passing values back from a function, and of using global variables, whose values are retained in all parts of a program.

The library.

The library is one of the glories of 'C', because it's the way in which the language can be continually extended and made more useful. The library is a set of functions, written in source code. Its value is that any function in the library can be named in your own program, and taken from disc or cassette to be included in your program. In many versions of 'C', this has to be done by loading in the whole of the library routines before you compile. HiSoft C has a very useful variation on library use which allows you to load in only what you need. This means that the disc or cassette has to be searched, and this takes a noticeable time, even for a disc. As with any compiled language, however, the time you spend on this part saves time later when you run the program. At this stage, we're not ready to make the most efficient use of the several library files that HiSoft has provided, but it's a good point to start looking at how the library is used.

To start with, there are two really important library files that you need to know about. The first one is called **stdio.h**, and the second is called **stdio.lib**. You will normally need both of them, and you certainly can't use **stdio.lib** unless you already have **stdio.h** in place. The positions in which these are called are also important. The header **stdio.h** must be placed right at the start of a program, before anything else, even before the **#define** lines. The header is installed by typing **#include stdio.h**. The use of **#include** with a filename like this causes the compiler to look on the disc (or cassette) for the routines. The other set, **stdio.lib** goes right at the end of all your program sections. Now you will usually want to take only what you need from this one, so its installed by typing **#include ?stdio.lib?**. The question marks are a feature of HiSoft C which you will not find on other versions, and their use for selective inclusion saves a lot of valuable memory, a feature that you'll appreciate if you are using the older Amstrad machines like the CPC464 or CPC664.

The next step is to decide what to use from the library. One useful function is the one that provides the poke action, and its syntax is almost identical to the use of POKE in BASIC. On the V1.2 version of the HiSoft compiler, poking location 3551 with 1 will switch to Mode 1, making the load on your eyes a lot easier.

```
/* Set mode 1 */
#include stdio.h
main()
{
poke(3551,1);
}
#include ?stdio.lib?
```

A program which sets Mode 1, using the library. This will remain set until you switch off, or change the number stored in address 3551.

FIGURE 3.2

This has to be done with the aid of the library, however, so we start the program of Figure 3.2, after the remainder line, with `#include stdio.h`, and end it with `#include ?stdio.lib?`. The rest simply consists of a `main()` that has just the one statement, `poke(3551,1)`. Compiling this takes a considerable time, because the library has to be searched. Even on disc this takes time, and I would want to make a cup of coffee if I were waiting for a cassette to do this. When the program runs, you don't at first notice any change. That's because the mode number is not used until a 'c' is typed. When you type 'c', then (RETURN), you'll see the mode change to Mode 1. This will persist for as long as you use the compiler, unlike the change which was noted in Chapter 2. If you happen to know how to make a new recording of your version of the compiler, you can now record it with this change in place. If you don't know how to do this, then it's not my place to tell you!

That's just one example of the use of the library. It's a long winded one, because only one library function has been used after all that disc spinning. If you want to use just one library function, the easiest way is to print out the function that you need and simply copy the library routine into your main routine. This will then compile at once, with no need to include either the header or the main library. If you want to do this with the `poke` action, Figure 3.3 shows the result.

```
/* Set mode 1 */
main()
{
  typedef char * _char_ptr;
  int address,value;
  address=3551;
  value =1;
  *cast(_char_ptr) address = value;
}
```

The program for setting Mode 1 with its routine extracted from the library (courtesy of HiSoft). This makes compiling much quicker.

FIGURE 3.3

This is simplified, because only fixed values are needed – but don't ask how it works just at this point! Check that it works, then save the code under a filename like `mode1`, and use it each time you have loaded in 'C'.

Look now at another couple of library actions. These are `max` and `min`, and as the name suggests, they will weed out numbers from a list. The list must follow the `max` or `min` words, within brackets, and the result of the search has to be an integer number. The `max` and `min` functions are in the `stdio.h` part, and if you don't need to use any of the functions from the main library, then you needn't run it. Figure 3.4 shows `max` and `min` used in a simple example.


```

#include stdio.h
main()
{
int k;
k=max(23,1,45,67,22,34);
printf("\n Max is %d",k);
k=min(23,1,45,67,22,34);
printf("\n min is %d",k);
}

```

Using the **max** and **min** functions from the **stdio.h** library.

FIGURE 3.4

You can, incidentally, speed up the compiling of this and other examples once you have checked that they operate correctly. If you put **#list-** at the start of a program, it will suppress any listing. You can see this command being used when the **stdio.h** program is being loaded in. Placing **#list+** at the end of the program turns the listing on again so that you can use the editor normally. This avoids having to watch all the titles come up in the library routines, and it's particularly useful for long programs.

As a last example of the use of the library for the moment, take a look at Figure 3.5.

```

#include stdio.h
#define sample "123fg"
main()
{
int val;
val=atoi(sample);
printf("\nValue is %d",val);
}
#include stdio.lib

```

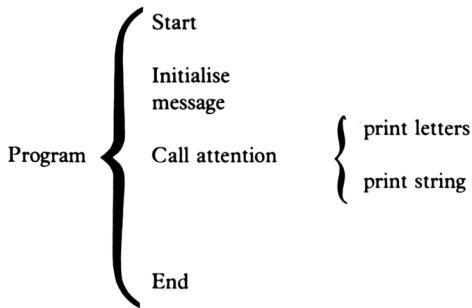
Using the function **atoi** from the library. This carries out the action of **VAL** in **BASIC**.

FIGURE 3.5

This defines a string constant as "123fg". The **stdio** routines are used, and the particular routine here is a very useful one, **atoi**, which does the job that **VAL** does in **BASIC**. This is to convert a string into an integer number. Only the number characters at the start of the string are converted, just like the **VAL** action. In this example, the string is a string constant, and the **atoi** action converts it into an integer 123, which is printed in the usual way. There are many more functions which will act on strings, but before we can make really effective use of them, we need to be able to work with pointers, –the crowning glory of 'C'. That's for later!

Planning.

One of the most important points about functions is how they affect planning of programs. How, for example, do we plan the simple illustration of Figure 3.1? Figure 3.6 illustrates a version of one method which is favoured by many programmers.



A planning system that is very popular with programmers.

FIGURE 3.6

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

Some operations.

There aren't many programs that you are likely to write that don't involve the operators of 'C'. The operators are the symbols which control actions on numbers, and 'C' is rather richer in operators than BASIC. In addition, some of the actions and the order in which they are carried out need rather more thought than you would give to similar things in BASIC. The four main operators of * / + - are specified just as they are in BASIC, but you have to remember the restriction of integer numbers in HiSoft C V1.2. Look at an example to start with, in Figure 3.7.

```

main()
{
int w,x,y,z;
x=3;
y=5;
z=7;
printf("\n%d divided by %d gives %d and
%d over",z,x,z/x,z%x);
w=++x;w=++w;
printf("\nw is %d and x is %d",w,x);
}

```

Integer arithmetic, showing the whole number result and the remainder or modulus.

FIGURE 3.7

This shows four integer variables declared, with three of them assigned. The first **printf** line will print values of *z*, *x*, *z/x* and *z%x*. Now *z/x* is just *z* divided by *x*, as you would expect, but this is **integer** division. Hisoft-C in its V1.2 form does not support ‘floats’, meaning numbers which contain fractions, so only the whole number part of 7/3 is printed, and that’s 2. The % operator, however, is used to find a remainder. The expression *z%x* means ‘find the remainder after *z* has been divided by *x*’, and this amount will be 1. The use of these two operators, then, allows you to carry out divisions and show the result as a number and a (vulgar) fraction, whereas using a float number would allow the result to be shown as a number and decimal fraction. Try altering the **printf** line so that the program will produce:

7 divided by 3 gives 2-1/3

– this is a good way of checking that you have really understood how **printf** works!

The next line carries out an action which is quite certainly unfamiliar in BASIC. The assignment **w=++x** means that the value of *x* is incremented, and it is then assigned to variable *w*. The next part of the line then increments this value of *w* and assigns it to *w*. The printed values of *x* and *w* then show that has been done. The ++ operator means increment (increase by 1) and the -- sign means decrement (decrease by 1). Just as important, however, is the point that the **position** of these symbols is important. Edit the assignment line now so that it reads:

w=x++; w=++w;

and run this. The printout now states that *w* is 4 and *x* is 4. Using the increment sign **following** the variable name means that the increment action has been carried out after the assignment, not before. If you make the line read:

w=x++;w=w++;

then the result is 'w is 3 and x is 4', because `w=x++` made `w=3`, and then `x=4`, while `w=w++` made `w` equal to `3`, and then carried out an increment action *which had no effect because the variable had already been assigned*. The use of increment and decrement can be very handy in loops, but you have to think out the order of things carefully. This becomes more difficult when you get to complicated expressions, so it's advisable to start with the easy ones.

Operator	Action
++	Increment variable value.
--	Decrement variable value.
*	Multiply quantities.
/	Divide (integer result).
%	Modulus, remainder of division.
+	Add quantities.
-	Subtract quantities.
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
==	Identical to.
!=	Not equal to.
&&	AND action.
	OR action.
?:	Select one or other.
=	Assign value.

The ordinary arithmetic and logic operators. See Chapter 10 for the bit operators.

FIGURE 3.8

Figure 3.8 gives a list of the ordinary operator symbols that are used in arithmetic or logic, and their actions. This is not a complete list of all operators, because there are operators which act on pointers, and others which operate on the bits of a byte, but these don't concern us at the moment.

```

main()
{
int a,b,w,x,y,z;
x=3;y=5;z=7;
test(x,y);
test(x,z);
if (y!=z) printf("\n%d not equal to %d",
y,z);
x*=6;
printf("\nx is now %d",x);
}
test(a,b)
int a,b;
{
if (a==b) printf("\n%d equals %d",a,b);
else printf("\n%d is not equal to %d",a,
b);
}

```

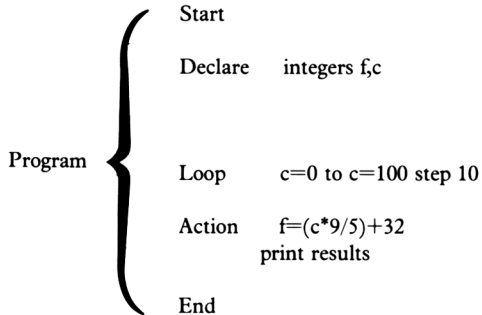
Examples of use of some operators.

FIGURE 3.9

Figure 3.9 shows another example of some of the operators which will be less familiar to you, along with an introduction to the 'if' test. The usual declarations of variables are made, then assignments, and then a function is used to compare first x and y, then x and z. In this function, **test**, the numbers which are to be compared are supplied in variable form as arguments in the brackets, separated by a comma. The **test** function itself uses integers a and b, which are allocated with values only in the function. We have declared a and b in the main program also, so that we could, if we liked, use a and b to hold quite different numbers in the main program, unaffected by the use of a and b in **test()**. The test is for equality, and the operator which is used is **==**, not the more familiar **=** of BASIC. In 'C', **=** is used for assignment, and **==** means 'identical to', avoiding the confusion that can arise in BASIC, which uses the same symbol for both purposes. The **test** routine will print a message if the two quantities are equal (which they never are), and another message if they are not equal. Notice how this second message is obtained. The **if** test line is followed by another line which starts with **else**, and this is followed by the action which is to be taken when the first test turns out to be false. The item which is to be tested is enclosed in brackets, like **(a==b)**, and this expression will be either TRUE or FALSE. If it is TRUE, then whatever immediately follows the test will be carried out. If the expression is FALSE, the program moves to the next line to carry out the effect of an **else** if one exists. If there is no **else**, then whatever follows the **if** line will be executed. You can have complicated sets of **if** and **else** lines with nesting, but for the moment we'll concentrate on the simple examples. When these tests have been carried out, there is another test in the main program which tests for **(y!=z)**. This means **y not equal to z**, and the **!=** combination in 'C' is the equivalent of **<>** in BASIC. Finally comes the expression that is decidedly peculiar to 'C'. Using **x*=6** is the equivalent of **x=x*6**, a combination of the operation of multiplication and an assignment. This kind of shorthand action is what gives 'C' a reputation for being difficult to follow. The point is that you aren't forced to use **x*=6** in place of **x=x*6**, but when you're familiar with these shortcuts these are very convenient.

More planning.

Let's look now at a short program, starting with the design steps. This is to be a program which will convert a list of Celsius temperatures into their Fahrenheit equivalents. The planning, such as it is, is shown in Figure 3.10.



Planning a simple program to convert Celsius temperatures into Fahrenheit.

FIGURE 3.10

On the left-hand side, the main steps of the program are listed as Start, declare, loop, action and End. The curly brackets now open into more detail. The variables will be called *f* and *c* and will be integer numbers (not much choice with V1.2). The range and steps will be dealt with by using a loop, which is something new for us. The only detail which is included in this list is the range of temperatures, 0 to 100 in steps of 10 Celsius degrees. Fortunately, this range allows exact conversions, so that we don't have to show fractions of a degree, but if we did, then you know how to do this. The conversion could be done by a function, but it's so simple that it's hardly worth while, and we simply use the conversion formula.

Now this program illustrates that a FOR loop in 'C' takes a very different form, particularly as regards the STEP portion. Figure 3.11 shows the program which has been drawn up from the plan.

```
main()
{
  int f,c;
  for (c=0; c<=100; c=c+10)
  {
    f=(c*9/5)+32;
    printf("\n%d C is %d F",c,f);
  }
}
```

The Celsius to Fahrenheit program for a range of temperatures.

FIGURE 3.11

As always, looking at a finished program gives you no idea of what the steps were in writing it, so we'll look at the program in the order in which it was written. When you write a 'C' program from a plan, never allocate any line numbers. Leave the allocation of line numbers to the *i* action of the compiler when you enter the program. This avoids continual re-numbering when you have to squeeze yet another definition into the start of the program at the planning stages! The main program starts in the usual way with `main()`, and then declares the integers `f` and `c`. The next step is the loop which carries out the actions of converting and printing the values. Now the first thing to note here is that the loop line *does not end with a semicolon*. This is because the statement has not ended; we have to specify what will be done in the loop, and that follows enclosed in curly brackets. The effect of enclosing the two statements in curly brackets is to make this set of lines constitute one single statement. A set like this is often called a 'compound statement', and because it ends with a closing curly bracket, it doesn't need a semicolon. What is inside this set of curly brackets, then, will be carried out on each pass through the loop.

The next thing to look at very carefully is how the loop statement is constructed. In many ways, this corresponds exactly to the BASIC statement:

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

but 'C' writes this as a set of conditions. The first test is `c=0`, the starting condition for the loop. The next is `c<=100`, meaning `c` less than 100 or equal to 100. This is the ending condition. The third is `c=c+10`, and this is the stepping condition. If you think that is all very clear and straightforward, then try omitting the step condition. You'll find that the loop is then endless, and you need to use ESC to get out of it. Unless you put in a step condition, there won't be one, and the loop will be endless, unless `c` gets incremented somewhere within the curly brackets following the loop statement. The other difference from BASIC is the condition `c<=100`. If you try making this `c=100`, you'll find the loop goes only as far as 90, because after 90, `c` is not less than 100. Now try making the middle condition `c=100`, and see what this does. The effect, another endless loop, is most unexpected if you are still thinking in BASIC terms. The reason is that *each part of the loop statement is a condition*. The `c=0` part is a starting condition. The `c<=100` is an ending condition, but the loop does not end until this condition is FALSE. If you put in a condition which makes the loop impossible, the result is an endless loop. With `c=0` (and `c=100`) false, the loop should not run, and the response is to increment `c` to 100, then run continuously! The conditions `c<number`, `c>number`, `c<= number`, or `c>=number` are the ones that you should always use for a loop of this type. Since 'C' offers you the choice of two other loops, you can't really complain that you are restricted for choice.

The conversion to Fahrenheit uses the variable name `f`, and is the first action in the loop. The next action is to print both amounts. With the range of quantities that we have chosen, there will never be a fractional result, so that the answers are exact. The way that the numbers are presented, however, could be improved. One way is to use left-justification, and this can be done by using the line:

```
printf("\ n%-3d C is %d F",c,f);
```

which lines up the printing of the words. This isn't perfect, though, because we don't normally left-justify numbers. A better display is obtained by using:

```
printf("\n   %3d C is %d F",c,f);
```

which looks a lot better. Three spaces have been typed between 'n' and '%', and the number has been right-justified to three figures. This puts the number always hard against the right side of the spaces that you have left for it, and makes the display look just right.

While we have a loop operating, we can take the chance to make some changes which will illustrate a few more points about how 'C' uses loops. In particular, 'C' has statements that allow you to skip passes through the loop, or to break out of the loop, without giving the computer's operating system apoplexy. BASIC is not nearly so well organised in this respect. Take a look at Fig.3.12.

```
main()
{
  int f,c;
  for (c=0; c<=100; c=c+10)
  {
    if (c==20) continue;
    if (c==70) break;
    f=(c*9/5)+32;
    printf("\n   %3d C is %d F",c,f);
  }
}
```

The use of **break** and **continue** in a loop.

FIGURE 3.12

Two lines have been added within the loop this time, and when you run the program you'll see that each has a very interesting effect. The **continue** statement makes certain that nothing more is done, the program action returns to the loop step for the next pass. In this example, no calculation is made, and nothing is printed. This is a way of excepting certain items (even-numbers, short names, one particular name) from being treated by the action of a loop. You can choose your position for the **continue**, too, because you might want to do some of the loop actions before you skipped the rest. The other loop modifier is **break**. This, as you might expect, allows you to break out of the loop altogether as the result of some test. In this example, it's when the value of *c* reaches 70, so that the loop prints out only as far as 60C. In this example, of course, it makes little sense to do this, because we could just as easily have put this in as the ending condition at the start of the loop. You might, however, want to test for another condition, such as something non-numerical, in this way. If your loop was, for example, reading a list of 100 names, you might possibly want to stop when you found McTavish, knowing that this name could be anywhere in the list. The **continue** and **break** actions of 'C' avoid the messy and unpredictable effects of using GOTO's in BASIC loops.

More functions.

It's time to take another look at a function action, one which is simple but rewarding to consider. This time, as the listing of Figure 3.13 shows, the function is called (or *invoked*) as part of a **printf** statement.

```
main()
{
  int x;
  for (x=0; x<=20; x++)
    printf("\n%d squared is %d",x,square(x))
    ;
}
square(a)
int a;
{
  return(a*a);
}
```

A function which returns a number to the main program.

FIGURE 3.13

The other important point which this program illustrates is one way in which a value can be passed back from a function. This is not quite as straightforward as you might think, because in the examples we have used so far, any quantity that is to be passed to the function has to be defined in the function. If it is defined in the function, however, its value is lost when the function ends! All of the functions that we have used so far have printed out whatever value they calculated. In this sense, we have been using the functions in the way that other languages use procedures, a way of doing some action rather than a way of returning a variable value.

The illustration in Figure 3.13 is quite different. The loop makes use of numbers from 0 to 20, and the incrementing is taken care of by using **x++** as the third term in the **for** statement. Since there is only one statement in the loop, it can follow the **for** part, and be terminated with a semicolon which now marks the end of the loop. If you put the semicolon after the **for** statement you'll get a loop, but with nothing in it! The **printf** statement prints the value of **x**, and also the value of **square(x)**. Now **square(x)** is a function, so we have to define it somewhere, but the main point is how it comes to have a number value.

The answer is in the line **return(a*a)**; This means that a value is to be given to the function, and that value is **a*a**, the square of a number which has been assigned to variable **a**. Remember that **a** is local to the function, it has no value in the main program. The value of **x** is passed to **a** when the function starts, and the function therefore acquires the value of **x*x** at the **return** step – there is no change to the value of **x**. As an exercise, you might like to tidy up the screen presentation of this little lot. Incidentally, if you have **return;** as a statement by itself in a function, this is where the function will stop and return, but in this case without a value. It's the equivalent of the way that a BASIC subroutine uses **RETURN**.

Now you need not feel that you *must* pass variable values to a function in this way. You could just as easily use `x` as the variable all along the way, as the slight modifications in Figure 3.14 show.

```
main()
{
  int x;
  for (x=0; x<=20; x++)
    printf("\n%d squared is %d",x,square(x))
    ;
  printf("\n x is %d",x);
}
square(x)
{
  return(x*x);
}
```

Using a global variable in the function.

FIGURE 3.14

In this case, variable `x` has been used for both actions, both in the main program and in the function. Because `x` has a value in the main program, both the name `x` and its value can be passed to the function to be used. You cannot, however, omit this passing step. The value of `x` itself is not altered by the program, as the final printed value shows. You can't use `square()` in the `printf` statement or in the definition of the function. If you omit the (`x`) part, then `x` is not recognised in the function. Your compilation will then stop with the error message of **ERROR 37 undefined variable**. You may have declared `x` in the main program, but each function is a rule unto itself and doesn't recognise any meaning to `x` unless it is passed on, or declared locally. If you try to get round this by keeping the `square()` use and adding `int x;` within the brackets of the function, you are in for more trouble. The program will now compile, but what you see when it prints out is simply gibberish numbers. Declaring `x` in the function makes a local `x` available, and it doesn't get assigned with a value. What you see when you print its square is just what happens when the contents of memory that this variable uses is squared. This is a feature of 'C' that you need to be careful about. You can do things like this which are silly, but which follow the rules of the compiler. Because they follow the rules of the compiler, they are accepted and compiled, but when they run, the answers are garbage. Unless you *know* that the answers are garbage, though, you might not notice. It's important, whatever your state of expertise, to test programs with items that can be verified reasonably easily – but you ought to know that already.

Suppose that you want to return several values from a function. The **return** statement does not cater for this, and there's no simple answer at the moment. In other varieties of 'C' you can use variables that have been declared as **extern** which you can pass to and fro as much as you like, with values changed as you want. HiSoft C does not use **extern** in this sense, and you have to live with it. As it happens, you very seldom need to pass back a set of values. If you are printing numbers, they can be printed from within the function. If the numbers have to be used in a calculation, then the calculation can be done inside the function and only the result passed back. As it happens, there *are* ways, but they depend on these mysterious pointers that you keep hearing about. Perhaps it's time we took a look at these.

Chapter 4

Pointers.

A pointer is a type of number variable, and the reason for its name is that it 'points' to where something is stored. For example, suppose that you have the character 'C' stored in the memory of the Amstrad. What this actually means is that one of the memory cells is storing the number which is the ASCII code for 'C', the number 67. Now memory for a computer is organised so that each unit (or **byte**) is numbered, and we might know that the number of the byte which held our 'C' was 41967. This number of 41967, then, is the number which is the pointer for 'C'. We could, if we liked, store the number 41967 somewhere so as to make it possible for the computer to find where 'C' was stored, and this is precisely what the action of a pointer is. It would be rather a waste to store a pointer for every character, but we don't need to. All we need to do is to store a pointer to the **start** of any variable. Once we know where the start is, we can locate it and read the required number of bytes of data. This is something that is used a lot in assembly language programming, but seldom occurs in BASIC. A few machines, notably the MSX machines, have a BASIC function called VARPTR which comes back with a number that tells you where about in the memory a variable is stored. The Amstrad machines achieve the same effect with the use of @ preceding a variable name. In BASIC, however, there is little use for this action, and not many programmers make use of it, or are even aware of it. In 'C', however, pointers are a way of storing variables and getting access to them. This is not just a useful feature of 'C', it's something that is central to the way that the language is constructed. Without pointers, you simply don't get very far with 'C'.

Take a look at a simple example just to get the taste of all this.

```
main()
{
char *p, ccr;
ccr='C';
p=&ccr;
printf("\n%c", *p);
printf("\n%d", p);
}
```

Using a pointer, in this example to point to a character. The pointer number must be declared *and* assigned.

FIGURE 4.1

Figure 4.1 is a program which declares two variables of type **char**. One of these variables is **ccr**, which is a straightforward variable name. The other, however, is referred to as ***p**. Now the asterisk, in this context, means 'contents of'. What it implies is that **p** is an address in the memory, and a character can be held at that address. The program assigns the variable **ccr** with 'C', a single ASCII code, and then assigns the pointer by using **p=&ccr**. The **&** operator, used in this context, means 'address of'. The effect, then, is to store the character 'C' at the address held in the pointer variable **p**. The program then prints out the character, in the form ***p**, and the pointer value itself, which when I ran it on my machine gave -23569. Don't worry about the negative sign, it only arises from the way that numbers are converted. The actual address number that this corresponds to is 65536-23569, which is 41967. You would have seen this number printed if you had replaced **%d** by **%u** in the **printf** line.

A pointer in 'C' is a variable quantity which is the address of another variable. What makes the pointer valuable is that if the pointer is declared, the compiler does not need to have the other variable declared. For example, if the pointer to a real number is known, and is variable **p**, then the name of the real number does not have to appear earlier in the program. A pointer reserves space for a declared type of variable, what you put into the space later is your own business, provided that it's the correct variable type. In addition, the pointer is a number (unsigned), but what it points to can be any type of variable, simple (such as another integer) or structured (like a record which consists of a number of different types). We can then juggle with the pointers rather than with the variables themselves.

All of this sounds rather academic, so take a look at an example which reveals a little of what all this is about.

```
main()
{
  int x,y;
  x=5;
  power(x,&y);
  printf("\n %d  %d",x,y);
}

power(x,p)
int *p;
{
  *p=x*x*x;
}
```

Using a pointer to return a value from a function.

FIGURE 4.2

In Figure 4.2, two integers **x** and **y** are declared in the main program. The variable **x** is assigned with a value, but **y** is not. The function **power(x,&y)** is then called. The quantities that have been passed here are the variable **x** and its value, and the *pointer to y*.

The important item here is that we don't deal with **y**, simply its address pointer. In the function, the header declares that the content of pointer **p** is an integer, but we *don't need to declare p itself*. The value of **p** will be assigned as the address of **y**, but all this is implied rather than declared. We can then make the statement which assigns the contents of pointer **p** to the cube of number **x**, and the function ends there. Now if we had assigned $y=x*x*x$, then **y**, if it had been declared in the header of the function, would have been assigned this value, but it could not have passed it back unless you used the **return** statement. Using a pointer does allow a quantity to be returned in the form of its pointer address. The main program then prints a value of **y**, using the pointer address of **y**, which has now been changed by the function to give the cube of **x**.

Now this is strong stuff— the quantity that has been stored in a variable **y** has been changed without the need to have a line $y=x*x*x$, or even a direct reference to **&y**. All that has been done is to pass **&y**, the pointer to **y**, to the function. This is the way in which a function can return values to a main program, and it's a method that is very extensively used in the library functions. In order to make any substantial use of the library functions, we have to master this idea of using pointers. At the moment, one problem that has been hanging over us is how to enter numbers, so this seems a good time to introduce one way, the **scanf** function. This function is built into HiSoft C, so that we don't need to use the library to load it when we compile.

Now as it happens, **scanf** is not the easiest of functions to use, and a lot of programmers avoid it like the plague. The principles are reasonably straightforward, though, and it's principles that we want to look at. Function **scanf** is set out very much like **printf**, with a control section, and a list of the variables that you want to input. So far, it sounds just like good old BASIC INPUT A,B,C. The important difference is that **scanf** requires pointers to variables, not just variable names by themselves. The other thing, the one that causes a lot of frustration, is that **scanf** works to strict and rather old-fashioned rules, and can do the most amazing thing if you don't understand the rules.

```
main()
{
  int n,k;
  for (n=0; n<=10; n++)
  {
    printf("\n Number, please- ");
    scanf("%d%c",&k);
    k=k*k;
    printf("\n square is %d",k);
  }
}
```

Using **scanf** for the input of a number. The number must be assigned to a pointer.

FIGURE 4.3

Figure 4.3 illustrates a **scanf** action, and gives you a taste of its use of pointers and one of its peculiarities. The main program sets up a loop which will run from 0 to 10, 11 passes through the loop. In each loop, we want to print a brief message, input a number, calculate its square, and then print that value. The input of the number is the action that is assigned to **scanf**, and the syntax for this particular example is

```
scanf(“%d%c”,&k);
```

which at first sight looks rather baffling. Run it, and check that it does as it ought to, remembering that all the arithmetic is integer, so that squaring large numbers will give very peculiar results. On the whole, it does as you might expect, though you’ll notice that there has been an extra blank line. This is because of the use of (ENTER) to terminate the **scanf** line. You can miss out the `\n` portion of the second **printf** statement if you want to close it all up. So far, so good, but what’s the `%c` for in the **scanf** statement? You’ll see if you try the program with this removed. The first number is accepted, but following that one, the loop cycles round without waiting for you to enter anything! The reason is that you used the (ENTER) key after typing the number. Without the `%c` in the **scanf** specification, the (ENTER) character is stored and used each time **scanf** comes round. Since you don’t have time to enter a number, this isn’t done. Unless you are working with a loop, this action is of no importance, and a lot of books on ‘C’ don’t even mention it. By adding the `%c` into the ‘specifier’ part of **scanf**, you allow for the (ENTER) character, and the loop works correctly. There is another way of getting round the problem, which is to leave a space ahead of the `%d` specifier, and we’ll illustrate that method later.

There are a lot of possibilities here, but the important point to look at is how **scanf** deals with the address pointer to variable **k**. The quantity that is called for in **scanf** is `&k`, the pointer to **k**. The action of **scanf** is to assign the number that you type into this pointer, so that the variable **k** can be used with this value. It’s a very good illustration of a function being used to work with a pointer, and **scanf** is a function which requires that all of its returned values should be pointers. We’ll come back to **scanf** later, but for the moment try editing the **scanf** line so that the specifier part reads `“%d%c*%c”`. The `*%c` part makes the **scanf** action skip a character, and its effect in this case is to allow you to enter a number, but to hang up when you press (RETURN), and wait until you press (RETURN) again.

Arrays.

In BASIC, you have simple variables, such as integers, reals and strings; and you also have one structured variable, the array. By ‘structured’, I mean that an array name like **A** means not just a single value, but a set of values which carry distinguishing names like **A(1)**, **A(2)** and so on. ‘C’ is very well equipped with ‘structured’ variables, or structured data types, as they are called. One of these types is the array, and for a number of reasons we need now to look at what it is and what we can achieve with it. As you might expect, an array has to be declared at the start of a program and this declaration will include a name (the identifier), the number of elements in the array, and the type of data that is to be stored. This is just

what you would expect from our experience of arrays in BASIC. When you use a DIM statement, such as DIM Name\$(20), in BASIC, you are specifying the name **Name**, the type (string) and the number of elements (from 0 to 20, a total of 21). The main difference in 'C' then will be the way in which an array is defined rather than the information which is used.

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

```
int classmarks[20];
```

This provides the name of the array, which is **classmarks**, the number of items (20 of them) and the fact that each item will be an integer. This variable declaration can be made along with other declarations of integers in the same line. One important point to note here is the use of the **square** brackets. If you forget that you are writing 'C' and not BASIC, it's easy to refer to an item as **classmarks(12)**, when you should be using **classmarks[12]** instead. The error message that you get when you do something like this will not necessarily remind you of what has gone wrong. Since the array in this example is an array of integers, you can use **scanf** to get each item of the array. One very important difference between the BASIC array and the 'C' array, however, lies in the way that items are numbered. When you define a BASIC array as **A(20)**, then this allows for 21 items, **A(0)** to **A(20)**. By contrast, the 'C' array allows for just the 20 that you specified, and these will be [0] to [19], there will be no [20]. This might not stop you trying to use an item [20], and this is one of the things that you have to be very careful about because 'C' doesn't always stop you from doing foolish things, it lets you go ahead and pour garbage into the memory which you don't find until later!

```
main()
{
int num,classmarks[20];
for (num=0; num<=19; num++)
{
printf("\nMark %d - ",num+1);
scanf("%2d%c",&classmarks[num]);
}
printf("\f\n%25s\n","CLASSMARKS");
for (num=0; num<=19; num++)
{printf("\nItem %d got %d marks",num+1,
classmarks[num]);
}
}
```

Using an integer array. The array numbers are read in from the keyboard, and then printed.

FIGURE 4.4

Figure 4.4 shows an example of this array in use, and also some more formatting. In this program, a set of twenty marks is obtained. There's nothing to stop you from entering numbers like 5000, but the program assumes that the items will be between 0 and 99. Later on, we'll see that this can be checked in better ways. The items are entered in a loop, using principles that should be familiar by now. The prompt line uses `num+1` rather than `num` so that you can have numbers from 1 to 20 instead of 0 to 19. In the `scanf` line, the `%2d` allocates the numbers in twos. This is not ideal, because it means that if you type a four-figure number, it will be allocated to two sets of marks! For the moment, though, it will serve to keep the numbers below 99. The other point about the use of `scanf` here is that the array pointer is used directly, as `&classmarks[num]`. You don't have to use any intermediate integer here, as you are required to do in some languages.

When all of the items have been entered, the screen clears and the title **CLASSMARKS** is printed in the centre. This is done in the `printf` line by using the control string "`\ f \ n%25s \ n`". The `\ f` part clears the screen, and the `\ n` part takes a line down. The next part, the `%25s`, is a string 'field size' number for the word **CLASSMARKS**. The field size number represents the *total* size of string that is printed, and if the word is less than this size, it is padded with blanks at the left hand side (in other words, it is right justified). By using a positive number, we force the word to be printed with any of these padding blanks on the left-hand side. The choice of the number 25 with **CLASSMARKS** (which has ten letters) means that 15 spaces will be printed to the left of the name. Figure 4.5 shows how this can be used to centre any name.

1. Count the number of letters in the title, for example, 16.
2. Add this number to the number of screen characters per line, for example 40 in Mode 1 (example gives 56).
3. Divide this number by two, and use it in the field size. For example, "`%28s`".

How to centre any title, using the fielding number.

FIGURE 4.5

If, incidentally, a negative 'field' number is used, the excess spaces are printed to the **right** of the name. This can be useful for spacing the next name, but is not so useful for a heading.

The items of the array are then printed out in order, using the variable name `num` as the array number and `num+1` as the item number. The printing line has not attempted to make the spacing such that the lines will be uniform for both single-digit and two-digit numbers. This, once again, is a good exercise for you in fielding these printouts. If you have been used to the action of **TAB** in **BASIC**, the use of the field numbers can often be difficult to adapt to, and the more experience you have the better. The rule is to use a positive field number when you want to start a word away from the left-hand side of the screen, and a negative number to space the **next** item along so that it will fit neatly.

The method of using **scanf** with the number specified by **%2d** makes sure that no number of more than two digits can be entered. This is not always a desirable method of checking, however. The main problem is that if your finger slips and you enter 999 instead of 99, then 99 gets entered in one mark, and the last '9' becomes the first digit of the *next* mark. The program as it stands gives you no chance to do anything other than grin and bear it. This is a compiled program, remember, so you can't just use the old BASIC trick of commanding a GOTO to get you back into the right part of the program. This method of specification, then, is suitable only when there is no loop involved, so that it doesn't matter if something gets left over. If you want to be able to correct an item **without** running the program all over again, then an IF test (like BASIC) is a preferable method, and we'll look at another method shortly. There's no reason why the result should not then be assigned to an array which has restricted values. The golden rule is that your program should never bomb out when the unlucky user (you, perhaps) has just entered a lot of data.

Strings at last.

A string in HiSoft C can be regarded as an array of ASCII characters, ending with a zero. This is true in all other versions of 'C', (and in some types of BASIC) but in BASIC there is a ready-made string variable type, marked with the dollar sign, as one of the main variable types. HiSoft C, in common with other varieties of 'C', does not have these string variables ready-made; we have to define them for ourselves. A string is an array of characters and we always define it in that way, as for example **string[20]**. When you write characters for a string in a program you enclose them with quotes just as you would in BASIC, and you don't have to type in the zero that is always used to end the string. As an example, which is also a very important guide to the use of pointers, take a look at Figure 4.6.

```
main()
{
  static char st[]="EXAMPLE";
  int n;
  char *p;
  p=st;
  for (n=0; n<=9; n++)
    printf("\n%c, code %d, is in %d",*(p+n),
          *(p+n),p+n);
}
```

A string variable, and its use with a pointer to print out characters, codes and addresses.

FIGURE 4.6

Now in this example a string is assigned, and the assignment is different from what you might expect. Both declaration and assignment have been carried out at the same time, and this is possible only with a **static** variable in any variety of 'C'. This is not a restriction of HiSoft C because we are initialising an array, and no version of 'C' allows an **auto** array to be initialised. In addition, though, the number of characters in the string has not been declared; there is no number between the square brackets. This is something that can be done only in a combined declaration and assignment, and you can't split this into two statements like **char st[]; st[]="EXAMPLE"**.

The real meat of this example, however, lies in the use of a pointer defined as **p**. Now in the 6th line of the program, we make the assignment **p=st**, which looks very peculiar. It rather looks as if we are assigning a pointer, which is an address number, to the name for an array. You would expect by now that an assignment of this kind should be written as **p=&st[0]**, and it would make not the slightest difference to the way that the program works if you did so. This is another of these short-cuts of 'C'. The name of any array (and a few other compound data types, as we'll see) is the pointer address of the first item in the array. The array is stored in consecutive addresses in the memory, as the example shows, and the use of pointers is particularly handy just because of that. You don't, of course, have to make use of pointers if you just want to print a string or select one item from it. You can print a string by using a library routine, knowing, as you do now, that the pointer for the string is represented by its name. You can pick a letter from the string by using the fact that it is an array, so that **st[4]** is the fifth letter (the count starts at 0, remember). Later, we'll see that it is possible to assign and use a string using only the pointer, with no string variable name at all.

```
main()
{
    static char st[]="EXAMPLE";
    printf("\n%s",st);
    printf("\n Fifth character is %c",st[4])
    ;
}
```

Picking out a character from a string, using its array number. This is much simpler than MID\$ in BASIC.

FIGURE 4.7

Figure 4.7 shows string selection more clearly. Once again the string is initialised, and the complete string is printed using a **printf** line. Only the array name, **st**, needs to be used here, and no square brackets are needed. In the following line, the fifth character in the string is printed by using **st[4]** – remember once again that counting starts with zero. Using the idea that each letter in an array can be located by using its subscript number, you can always assign one array to another variable name. For example if you have variables **st** and **string**, both of which are character array types, you can use a routine such as is shown in Figure 4.8 to make a copy of the array **st** into the array name **string**.

```
main()
{
    static char st[]="Sample String";
    char string[20];
    int n;
    for (n=0;n<=19;n++)
        string[n]=st[n];
    printf("\n name is %s",string);
}
```

Copying one string to another – you can't equate the string names as you do in BASIC.

FIGURE 4.8

This is pretty much the same method as you would use in BASIC to copy one array into another. As it happens, you have a more efficient method available in the form of the `strcpy` function in the library.

```
#include <stdio.h>
main()
{
    static char st[]="Sample String";
    char string[20];
    strcpy(string,st);
    printf("\n name is %s",string);
}
#include <stdio.lib>
```

Using the library function `strcpy`.

FIGURE 4.9

Figure 4.9 shows how this can be included into a routine that needs a lot of library functions. Remember that if you need just one library function, it's much easier just to type the source code of the function that you want into your listing than to have to wait for the library to be read each time you compile. The library version makes use of pointers, as you might expect, and it's a good example of just how compact 'C' code can be made.

Let's get back to the strings, however.

```
main()
{
    char c,name[20];
    static int n=0;
    do
    {
        c=getchar();
        name[n]=c;
        n=n+1;
    }
    while(c!='\n');
    name[n-1]=0;
    n=0;
    while (name[n]!=0)
    {
        putchar (name[n]);
        n=n+1;
    }
}
```

Reading and writing a string character by character. This makes it easy to test for a 'terminator'.

FIGURE 4.10

Figure 4.10 shows a very simple string variable reading and writing program. This time, very different methods have been used for entering and writing the name. We have made use of some of the simple built-in functions that exist for reading and writing characters, and this has involved two different types of loop, the **while** and the **do..while**. The word **name** is defined as an array of 20 items of type **char**. This means that **name** should not contain more than 20 characters, but there is nothing to stop you from entering more than 20. This is a feature of 'C', that you have to build in your own safeguards, the language provides only the minimum necessary. If you enter only two characters from the keyboard no harm is done, but the rest of the array will be filled with garbage. If you enter 22 characters, the program may crash at a later stage. After the declaration and initialisation steps, the program starts a loop. This loop is not the kind that is regulated by a counting number, like the **for** type of loop. Instead, the actions that follow the keyword **do** are repeated until a condition that follows **while** (at the end of the loop) is TRUE. It's rather like the WHILE...WEND loop of the Amstrad turned the other way round. The test is made at the end of the loop, so that the loop must run at least once. The statements in the loop are enclosed in curly brackets, so that they act as one 'compound' statement. The **c=getchar()** function does what its name suggests, it gets a character from the keyboard. The next line assigns this character a place in the string, **name**, and following that, the counter integer **n** is incremented— I have used **n=n+1**, but you know a better method, don't you? The **while** condition is for the character to be the carriage return, indicated as '**\n**'— and note the apostrophe signs, not quotes, because this is a single character, not a string with a zero at the end.

So far, so good. Everything you enter at the keyboard will be taken and put into the array **name**. Everything, that is, including the **\n** at the end, but with no final zero, because we can't type a zero character (the 0 on the keyboard is ASCII 48, not ASCII 0). The statement **name[n-1]=0** remedies this problem by stepping back to the **\n** character and substituting a zero. Now we can print the string. You could, of course, make use of **printf** to do this, but since we're looking at the simple functions, let's use **putchar**. This, like **getchar**, works on one character at a time. Also like **getchar**, it has to be instructed rather closely, and we have to start by setting the counter **n** to zero again, otherwise we can't print the correct characters. Now we use a **while** loop. Unlike the WHILE...WEND in Amstrad BASIC, the while loop in 'C' carries out a set of instructions which either end with a semicolon, or are enclosed by curly brackets. In this example, we've used the curly brackets, so that providing that **name[n]** is not a zero, **putchar** will place the character on the screen, and the number **n** will be incremented. When the zero is found, the action stops.

Now it works, but no seasoned 'C' programmer could possibly be happy with it. The whole principle of 'C' is that you ought to be able to do a lot with a very few instructions. In an example like this, it's preferable to make use of some of the library routines which use pointers, but just for the exercise, how could we go about slimming this down? The answer is to make use of a few of the 'C' shortcuts, and some of these are illustrated in Figure 4.11.

```

main()
{
char name[20];
static int n=0;
while((name[n++]=getchar())!='\n');
name[n]=0;
n=0;
while (name[n]!=0)
putchar (name[n++]);
}

```

The program of Figure 4.10 slimmed down or compacted.

FIGURE 4.11

This is the same program, using the same routines and methods, but with shortcuts. It takes ten lines instead of nineteen to write, so the slimming, while not exactly anorexia computerosa, is impressive. The main reduction is obtained by using the line:

```
while((name[n++]=getchar())!='\n');
```

which is the sort of thing that gives 'C' a bad reputation with academics. When you unravel it it's not quite so bad as it looks, and you'll soon learn to shrink lines down to this state. The way to unravel anything like this is to start at the innermost brackets. Within these you'll find: **name[n++]=getchar()** and this assigns the character from the keyboard to **name[n]**, and then increments **n**. This allows us to remove the **n=n+1** line from the old program, and also replaces the clumsy assignment to char **c** and then to **name[n]**. This complete part of the statement is enclosed in brackets, following which is **!='\n'**, testing to find if what is within the brackets is *not* equal to the (ENTER) key code. The whole of this is enclosed in brackets which follow the **while**. This has the effect of testing if the whole expression is TRUE or FALSE. If the key is not (ENTER), the expression is TRUE, and the **while** loop is carried out. In other words, the character is put into the string and the place number is incremented. When the **\n** code is found, the expression which follows **while** is FALSE, and that's the end of the loop. The whole of this **while** loop is in one line, marked by the semicolon at the end of the line. The two actions which follow are as in the old version, putting a zero at the end of the name and making **n** equal to zero again. Using **name[n]=0** actually causes the string to include a **\n** character, but the effect of using **name[n-1]** doesn't make much difference until the printout stage when it changes the number of spaces under the printed version. We can, in fact, save another line here by using:

```
n=name[n]=0;
```

which carries out the two assignments to zero in one step. Finally, the **putchar** part of the work is done in two lines with another **while** loop, and you should be able to unravel that one from your experience of the **getchar** loop.

In HiSoft C, string actions look rather complicated because assignment is not quite so easy as you might think. Since a string is a form of array, and there's no method by which you can copy one array into another except character by character, it all looks like hard work. It's really only a problem, however, if you are 'thinking in BASIC'. You can assign a **string constant**, for example, by using **#define** in a lot of places where in BASIC you would use a string **variable**. You can also use the scheme that we looked at earlier, of defining a **static char string[]** into which you can assign anything you want. You can also make a pointer point to any string you want, which is probably the easiest way of re-assigning a string name. We don't need to go into this, because it's in the HiSoft C library (it's one that exists in every 'C' library). When you really need to use a string variable is when you are inputting or outputting strings, and once again there is a library routine, **gets(string)**, for this purpose. We'll look at these routines later. The thing that you really have to be careful about is any attempt to assign a string to an array which is not large enough, because only a string of at least the same length is compatible. If your strings are arrays of 20 characters, then only another array of up to 20 characters *total* (including zero or any **\n** character) can be assigned. This is very difficult to get used to when you have been accustomed to the free and easy ways that BASIC has with strings, and Chapter 8 of this book is devoted to ways of making life easier for you.

Arrays of strings.

In BASIC, you are accustomed to being able to use string arrays, with assignments such as **A\$(5)="FRIDAY"**. In HiSoft C, a string array is an array **of an array** of characters. This sounds complicated until you realise that it's no more than a two-dimensional array. The sort of thing which in BASIC is written as **A(3,4)** is treated very similarly in HiSoft C, with the minor difference in this case that one of the dimensions is a set of ASCII codes. An easier method is to define a string as an array of characters, and then define another name as an array of strings. Once you have defined your 'string array' name correctly, you can use the string array rather as you do in BASIC. You **must** remember, however, that the rules are rather more strict. Each name in the array, for example, will consist of not more than the declared number of characters, and it's likely that anything following the zero that marks the end of a string will be garbage.

Look for example at Figure 4.12.

```
main()
{
char name[10][20];
int n,j;
for (n=0;n<=9;n++)
{
j=0;
printf("\n Name please - ");
while((name[n][j++]=getchar())!='\n');
name[n][j-1]=0;
}
printf("\f");
for (n=0;n<=9;n++)
printf("\n%s",name[n]);
}
```

Using a string array, a two-dimensional array of characters.

FIGURE 4.12

This consists of a program which will fill an array with names (of up to 20 characters), clear the screen, and then print the lot out. We start by defining **name** as an array of ten strings, each of which is an array of characters, up to 20 characters long. Two integers, **n** and **j**, are then defined to be used as counters. In the first loop, using **n=0 to n<=9** because the array elements are 0 to 9 not 1 to 10, the array **name** is filled with names that you type from the keyboard. Each string is referred to by its two numbers, the place in the array of strings, and the character in each string. For example, **name[2][4]** means character 4 in string 2, remembering that character 4 is the *fifth* character, and string 2 is the *third* string. The screen is then cleared, and the array of strings is printed on the screen by using the other loop. Each name in the array is obtained, once again, by using its position number within square brackets; **name[7]** in HiSoft C is equivalent to **NAME\$(7)** in BASIC. Note that this time you don't have to use two sets of square brackets. By specifying that you want to print a string, you have automatically made it unnecessary to specify the second set of square brackets. Just as you could use **printf** to print a string called **title**, which was defined as **title[25]**, you can use **printf** to print a string which is called, for example, **name[4]**.

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

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

so that any day can be found by use of the array number. This is not quite so easy in HiSoft C, because in 'C' there is no direct equivalent of the BASIC **READ** and **DATA**

instructions. These instructions are very wasteful of memory, because everything that you have in a DATA line in BASIC is stored in two places while the program is running. The action of READ....DATA in BASIC is really just the initialisation of an array in 'C', and the program of Figure 4.13 illustrates this action in a form of a function which you can use for your own programs.

```
main()
{
  static char *week[]={"Monday","Tuesday",
    "Wednesday","Thursday","Friday"};
  int n;
  for(n=0; n<=4;n++)
    printf("\n Day %d is %s",n+1,week[n]);
}
```

Initialising an array, the equivalent of using READ..DATA in BASIC.

FIGURE 4.13

This time the array is an **array of pointers**, one pointer for each string, with no restriction on string length; a subject that we'll come back to. The name of the pointer is **week[]**, it points to a character type, and its initialisation is carried out as shown. The storage class must be static if we are to carry out declaration and initialisation in one line, and the new feature is how a set of words, between quotes, can be put into an array. The contents of the array are shown between curly brackets, separated by commas. In an initialisation there is no need to show the number between the **square** brackets of the name, so this goes in as ***week[]**. When the array of strings is printed out, we don't print ***week[0]**, ***week[1]** and so on, but **week[0]**, **week[1]** etc. This is because the pointer name is the name of the array item. This is the kind of thing that's always likely to catch you out when you first start writing programs in 'C', and it's the first thing to suspect if you find that a printout gives you a screen full of gibberish. Note, incidentally, that each string will end correctly with a zero. You haven't put this in, but it's taken for granted when you use letters between quotes, like "Monday".

So far, we have been looking at comparatively short programs. When your programs get to the length at which they take up more than one screen 'page', a printer becomes a more pressing necessity. It's particularly useful if you are using pointers and you are not sure whether you should be using a ***x** or just **x** at any particular time. If you can see the declarations at the same time as you look at the lines that are giving you the problems, it all becomes much easier. Another problem is that by the nature of 'C', you tend to have a lot of nested { and } marks. If you can see these only on the screen, it's very difficult to be sure that each { corresponds to the correct }. Of course, if you planned the program correctly in the first place, you will have checked the nesting on paper. The problem arises, however, when you have been doing some editing, renumbering the lines, correcting mistakes and so on. At that stage, checking for an incorrectly placed } on the screen alone can be rather a frustrating task. One thing that can make a 'C' program much easier to read is indenting each new { or loop. In this way, sections which are 'compound statements', running as if they consisted of one single instruction, are set away from the left-hand side in a block. This makes it easier to see where the { and } of each block is located.

Chapter 5

Menus, choices and files.

The BASIC of the Amstrad allows the simplest method of programming menus, using the ON K GOSUB type of command. Since a menu is a very common feature of a lot of programs, it's time that we took a look at how such a system can be programmed in HiSoft C. The key to simple menu programming is the **switch** statement, which is 'C's equivalent of ON K GOSUB. Suppose that you have a list of items on your menu, with each item numbered in the usual way. You then use a keyboard read function to input a number. Suppose, for example, that you use the **getchar** function, which is built-in. You can then program:

```
switch(getchar()-48)
{
case 0:(first action);
break;
case 1:(second action);
break;
```

and so on, with the closing curly bracket at the end of the list. The function **getchar()** will have values which are ASCII codes for numbers such as 0,1,2, and so on, so that **getchar()-48** converts to number form as we saw earlier. This allows **switch** to select the command which appears after the same number in the list that follows **case**. In a real program, each of these options would consist of a function name, or a statement. For an illustration, we can substitute simple **printf** statements, as in Figure 5.1. The important point is to understand why the **break** statement has been added in each line.

```

main()
{
printf("\f%24s\n", "THE MENU");
printf("\n 1. Start file.");
printf("\n 2. Add to file.");
printf("\n 3. Delete item.");
printf("\n 4. Amend item.");
printf("\n 5. End program.\n");
switch (getchar()-48)
{
case 1:printf("\n file starts here.");
break;
case 2:printf("\n add to file here.");
break;
case 3:printf("\n delete item here.");
break;
case 4:printf("\n amend item here.");
break;
case 5:printf("\n end of program.");
break;
default:printf("\n No such item- please
try again.");
}
}

```

A skeleton menu program, showing how the **switch** statement is used.

FIGURE 5.1

In this example, the screen is cleared by the `\f` part of the first **printf** statement. This prints a title, and the fielding command `%24s` has been used. Note that when anything like this is done, the message must be separated by a comma from the fielding. If you use `printf("\f%24s\n THE MENU");` you will see a set of gibberish characters appear preceding THE MENU. The menu items are then printed, with a number allocated to each item. You are asked to choose by number, and then your number choice is put into the **switch** statement. What follows lists the numbers and actions. Each number is followed by a colon, then the action or actions that must be carried out. The set of **switch** actions must end with a closing curly bracket, and the whole program ends as usual with the final curly bracket. Each choice has simply caused a phrase to be printed in this example, because the aim is just to show what the **switch** statement does and how it is programmed. To see why we need the **break** statements, try omitting one or two. You'll see that this has the effect of allowing more than one answer to be printed. The **switch** statement allows you to select one item, but when the action returns, it will move to the *next case statement*. Unless you want the next **switch** line to be carried out, you must make this next statement the **break** to allow the rest of the **switch** sections to be skipped. Notice, too, that we can cater for a selection which is not in the range that **switch** allows. This is done by the **default** item, and it's a very handy way of ensuring that the entry range is checked and something sensible done for each possible answer.

In many examples, though, that's still not quite what we want. What we would like is to have the menu repeated until we enter a number that is suitable. In primitive varieties of BASIC you have to use a GOTO to achieve this, but in 'C', the obvious way is to use the **do..while** instruction. This is illustrated in Figure 5.2, in which the selection is repeated until a choice in the correct range is made.

```
main()
{
int j;
printf("\f%24s\n", "THE MENU");
printf("\n 1. Start file.");
printf("\n 2. Add to file.");
printf("\n 3. Delete item.");
printf("\n 4. Amend item.");
printf("\n 5. End program.\n");
do
{
switch (j=getchar()-48)
{
case 1:printf("\n file starts here.");
;
break;
case 2:printf("\n add to file here.");
;
break;
case 3:printf("\n delete item here.");
;
break;
case 4:printf("\n amend item here.");
break;
case 5:printf("\n end of program.");
break;
case -38:break;
default:printf("\n No such item-%d please try again.\n",j);
}
}
while(j>5 ||j<1);
}
```

Repeating the **switch** action until a number in the correct range is entered.

FIGURE 5.2

This time, the program does not end when an incorrect choice is made. The message is printed by the function, and the **do** loop ensures that the choice can be made again until the number lies in the correct range. It's not quite so straightforward as it seems, however. If

you simply add a **do..while** loop, you need a quantity to test at the end, and this can be obtained by using a variable to store the value obtained from **getchar**. In the example of Figure 5.2, the integer **j** has been used. You don't need to equate this to **getchar()** in a separate line, it can all be contained within the **switch** statement as the example shows. At the end of the loop, the value of **j** is tested. This contains the statement **(j>5 | | j<1)**, and the novelty here is the vertical bar signs. Used in this way, in pairs, they mean logical **OR**, so that the statement in brackets tests the truth of 'j greater than 5 OR j less than 1'.

The trouble is that this always causes the default message to be issued twice, once for the incorrect number, and once again for the (ENTER) key. This is because entry from the keyboard is done by storing the characters in a memory buffer. The (ENTER) key returns ASCII 10, and 10-48 gives -38, so this is assigned to **j** after the first default message, causing another message – but there are no more characters left now in the buffer. Now this could be sorted out by a bit of machine code which clears out (or flushes) the buffer, but there is a simpler 'all-C' solution, which is to make a **case -38:break;** to detect this and ignore it. This is one of the delightful things about 'C', that there is so often a way out of difficulties which doesn't involve digging into the machine-code. This is important, because the Amstrad machines are not so compatible with each other as you might think, and it's always best to avoid machine-code unless you are certain that it works on the machine that you are using.

Incidentally, now that we're starting to look at programs which contain several sets of nested curly brackets, it's time to think of indenting program lines. Indenting means leaving a space at the left-hand side, and it's a good way of showing how statements within curly brackets are nested. If you have nested sets of curly brackets in a program, put each new starting bracket one space in from the previous one, and indent all of the statements within the brackets similarly. This makes it much easier to read the program and see which statements are included in which set of brackets. The listing for Figure 5.2 should give you some flavour of all this. The semicolons at the end of the **switch** lines have spilled over because the printer has been set to use the same 40-character lines as the screen.

Other cases.

The control for **switch** does not have to be defined as an integer because, as we have seen previously, a character is entered as an ASCII code which is an integer anyhow. You can, therefore, use a character to control a **switch**, as is illustrated in Figure 5.3.

```

main()
{
char s;
s='@';
do
{
if (s!='@')s= getchar();
printf("\ntype a letter\n");
s=getchar();
switch(s)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':printf(" --is a vowel\n");
break;
default:printf(" --is a consonant\n");
}
}
while (s!='@');
}

```

Controlling **switch** with a character rather than with a number.

FIGURE 5.3

In this example, the variable **s** is of type **char**, meaning a letter, and the **switch** statements are set up for letter testing. We can still use **s=getchr()** to get the character from the keyboard, however. This is because, once again, the language does not make any rigid division between characters and integers – the main difference as far as the computer is concerned is that a character is stored in only one byte of memory, and an integer requires two bytes. Note that a character is referred to by using its key, within *single* quotes, such as 'S', 'A' and so on. This is something you constantly have to remember, because using double quotes, such as "S", "A", means a string which consists of the letter code *and a zero*. Once again, in this program, the use of a **do..while** loop will cause problems, in particular the repetition of the 'type a letter' message. This time, the method that has been used to prevent this is different. It is certainly possible to use an extra case line to detect the character '\ n', but this does not prevent the prompt from being printed twice. The test at the start of the loop, however, along with the assignment to '@' before the start of the loop does what is needed. Assigning the value '@' to **s** before the loop starts prevents the extra **s=getchar()** step from being used. If the loop returns because the @ key has not been pressed, the value of **s** cannot be @, so the extra **getchar** will read the (ENTER) code of '\ n', and allow the program to operate normally.

This type of character input can be improved by using some of the built-in library functions, and one of these can also be used to get over the (ENTER) key difficulties. The improved program is shown in Figure 5.4.

```

main()
{
    char s;
    printf("\nType a letter - use @ to stop\n");
    while ((s=getchar())!='@')
    {
        if (isspace(s)) continue;
        if (! isalpha(s))
        {
            printf("not a letter\n");
            continue;
        }
        tolower(s);
        switch(s)
        {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':printf(" -is a vowel\n");
                break;
            default:printf(" -is a consonant\n");
        }
    }
}

```

Analysing letters with a **switch** statement and some of the built-in character actions.

FIGURE 5.4

This time, the test for escaping from the loop (the @ character) is made at the start, using a **while** loop. You have to be careful how this is done, with the **s=getchar()** step enclosed in brackets and made not equal to '@', and the whole expression in brackets for the **while** statement. If you get these brackets wrong, such as by using **while(s=getchar()!='@')** then you will find that **s** is not assigned to any character that is not @, which is not exactly what you wanted. In the loop, two tests are then made. The first test uses the **isspace** function, which is TRUE if **s** happens to be a space, the newline character or a TAB. In this example, it's the newline we are trapping, and the effect will be to continue if the character is a newline. 'Continue' *used in any type of loop* means that the rest of the loop will be skipped, and the loop is restarted. If the newline is found, then, the loop returns for another **getchar**. The next test uses function **isalpha**. By using this in the form **if(! isalpha(s))**, we get a TRUE answer if the character is **not** alphabetical. For this event, we print out the 'not a letter' message, and continue to get another letter. If character **s** has survived so far, we then use function **tolower(s)** so that any upper-case letter is converted to lower case. This avoids the problem of entering an upper-case letter like A, E, I, O, U and being told that each is a consonant. All of these functions are built in, and don't need the library to be searched.

One last point about **switch**. The expression that follows **switch**, within brackets, must give a single integer. You can't for example, make **switch** work with strings, except to recognise the first character of a string. If you have to work with strings, then a program like the one in Figure 5.5 will be more suitable.

```
main()
{
  int j,n;
  char command[6];
  printf("\nPlease type command");
  printf("\nCLS,UP,DOWN,LEFT,RIGHT");
  do
  {
    n=0;
    while ((j=rawin())!=13)
    {
      command[n]=j;
      n=n+1;
    }
    command[n]='\0';
    if (!strcmp(command,"cls"))rawout('\14');
    if (!strcmp(command,"up")) rawout('\13');
    if (!strcmp(command,"down"))rawout('\12');rawout('\10');
    if (!strcmp(command,"left"))rawout('\10');
    if (!strcmp(command,"right"))rawout('\11');
    rawout(233);
  }
  while (command);
}

int strcmp(s,t)
char *s,*t;
{
  while (*s==*t)
  {
    if (!*s) return 0;
    ++s; ++t;
  }
  return *s-*t;
}
```

Using a menu structure to recognise command words.

FIGURE 5.5

The name **command** is defined as a string of up to six letters, and it is filled with characters by using a loop which contains **rawin**. This requires testing for the ASCII code of 13 to check the use of the (RETURN)/(ENTER) key. The point of using **rawin** in this example, is that it does not place anything on the screen. For some purposes, particularly graphics programs in which pressing an answer should not show on the screen, this action can be useful. The command word which is obtained is then compared with a list of keywords by using **if** tests with **strcmp**. *There is no way in which you can make a direct comparison of one string with another in 'C', so that lines such as:*

```
if (command=="cls")...
```

are not valid. The **strcmp** function, which is in the function library, does the comparison character by character, and returns a number whenever two characters are unequal. If the strings match perfectly, then the function returns 0. We have to test for *NOT strcmp*, therefore, using the **!** sign. The program example will do things like clear the screen and move a cursor about by using long commands such as UP, DOWN, LEFT, RIGHT. The important points are that the words do not appear, and that the comparison can be made. You will need to press the (ESC) key twice to get out of this one, because the **while** condition will make it loop forever.

Recording data.

Once you have made a start to gathering information into arrays, then it's likely that you'll want to record the information on to cassette or disc. The cassette filing system of the CPC464 follows the same command system as the disc system, and the later 664 and 6128 machines are disc-only, though a cassette system can be plugged in. In this part, then, we'll assume disc use, and if you use cassettes the only differences are that you'll have to choose places on the cassette and wait around rather a lot. We'll start at the beginning, and look at what is involved in recording and replaying a list of integers which will be held in the computer as an array. Figure 5.6 shows what is involved.

```
main()
{
  int n,a[51];
  int *fp;
  for (n=0;n<=50;n++)
  {
    a[n]=2*n;
  }
  printf("\nArray filled....");
  fp=fopen("intfil","w");
  for (n=0;n<=50;n++)
  {
    fprintf(fp,"5d\n",a[n]);
  }
  fclose(fp);
}
```

A disc or cassette filing program for a list of integers.

FIGURE 5.6

The integers are generated in a loop, which simply gives all the multiples of two up to 100. Once this array has been generated, the recording file is opened by using the line:

```
fp=fopen("intfil","w");
```

in which **fp** is a **pointer** to the start of the file, "**intfil**" is a filename that will be used on the disc, and "**w**" means write. Note that this is "**w**", a string, not '**w**', a character. The function **fopen** is built-in, so that you don't need to search the library for it. Once the file has been opened, the array can be recorded by using another loop, with a variation on **printf** being used in the writing process. The function **fprintf** is used very much like **printf**, but with the file-pointer as the first of its arguments. Once again, **fprintf** is a built-in function. The whole action could have been carried out in one loop, but I wanted to separate the generation of the numbers from the filing routine so that it would be easier to adapt the program for something more useful. The file must be closed by using **fclose(fp)** after writing. If there has been any other file called **intfil** on the disc, it will be renamed as **intfil.bak** in the usual way, and the new **intfil** file will become the current one of that name. Once the file is on disc, you can look at it, after a fashion, with the **type** command of CP/M. This involves losing the 'C' compiler to switch into CP/M, and typing **type intfil**. You will see the integers appear, very untidily, on the screen with a new line and a right-space for each new integer. To read the integers back in a more controlled way, we should write a reading program in 'C' – and that's the next step.

```
main()
{
int n,b[51],*fp;
fp=fopen("intfil","r");
for (n=0;n<=50;n++)
{
fscanf(fp,"%d\n",b+n);
}
fclose(fp);
for (n=0;n<=50;n++)
printf("%d ",b[n]);
}
```

A reading program for the disc/cassette file.

FIGURE 5.7

One possible reading program is illustrated in Figure 5.7. This one prepares in the usual way, and opens the file using **fp=fopen("intfil","r")** with the "**r**" (*not* '**r**') meaning "**read**" in this case. The loop is performed as before, but this time **fscanf** is used, and the syntax is not the same as that for **fprintf**. The reason is that an array is being filled, and **fscanf** needs a pointer to the position in the array. Now the name of the array, **b**, is the pointer to its first item, **b[0]**, so that if we use **b** by itself in the **scanf** instruction, all numbers will be read into the first item. To make the pointer shift to the correct item, we use **b+n**, so that the correct number of address bytes above the pointer start **b** is being used. When you add to a pointer in this way, the number that is actually added is a calculated number, taking into account the type of data. For example, an integer uses two bytes. If

pointer **b** happens to be 42000, for example for $n=0$, then for $n=1$ the address is 42002, because an integer takes two bytes. This automatic adjustment is very useful, but easily forgotten. After the numbers have been read, the file is closed in the usual way, and the array is then printed out. The printout is not in the same format as was used for reading the array in, which is the main benefit of using a separate loop for this purpose.

The use of **fprintf** and **fscanf** is just one of a set of ways of using disc filing.

```

#define EOF -1
main()
{
char c,a[51];
int n,j,*fp;
n=0;
fp=fopen("newchr","w");
c='\n';
while(n<=50)
{
if(c!='\n')
{
c=getchar();
continue;
}
printf("\nType a single character.. ");
j=getchar();
if(j=='\0')break;
c=j;
putc(c,fp);
n++;
}
fclose(fp);
printf("\nPress any key to read back");
while(!keyhit());
n=0;
fp=fopen("newchr","r");
if(fp==0)
printf("\nNo such file");
else
while((j=getc(fp))!=EOF)
{
a[n++]=j;
}
a[n]='\0';
fclose(fp);
printf("\n%s",a);
j=rawin();
}

```

Using **putc()** and **getc()** in filing programs.

FIGURE 5.8

Figure 5.8 demonstrates two other functions which can be used, **putc()** and **getc()**. As the names tell you, these are character functions, but this description can be very misleading, particularly as applied to **getc()**. The action of **getc()** is to return an **integer**, which can, of course, be regarded as a character in ASCII code. The important point is that you can assign **getc()** as an integer or directly as a character, but it's better always to assign it as an integer. The reason is that you generally use **getc()** in a loop which continues until the end-of-file character is found. The EOF character in HiSoft C is -1, which in integer form consists of two bytes. If you read **getc()** as a character, it will only deliver one byte, and the end of file character cannot be read. That's usually one fruitful cause of program crashes. Another is to gather the characters into a string and forget that there must be a zero at the end when the string is printed!

Looking at the program of Figure 5.8, then, the assignments are made as usual with character **c**, string **a**, and the others integers. As before, the pointer has been defined as pointer to an integer. The counter **n** is initialised to zero, and **c** to the newline character. This assignment has been made to avoid problems due to the newline trapping in the **while** loop that follows. In this loop, the number of characters that can be entered is limited to 51 (from 0 to 50), and a trap for a newline in the keyboard buffer is placed as the first step. After a letter has been typed, near the end of the loop, this trap will remove the stored newline character so that the message ('Type a single character') is not repeated unnecessarily. If **c** is not assigned to a newline before the loop starts, however, the message step will not appear until the (RETURN)/(ENTER) key is struck. The loop is also arranged to break when the '0' key is used – obviously you could use whatever terminator you wanted, and the message would normally mention this. Function **getchar()** is used to get the character from the keyboard, and the character is extracted the long way round, using integer **j**, whose value is tested, and then assigning this to character **c**. The file which was opened at the start of the program is then used by **putc(c,fp)** to place the character **c** in the file which is pointed to by **fp**. This loop continues until a '0' is entered or until the maximum permitted number of characters has been entered. The file is then closed, and the program hangs up, waiting for you to press a key.

The 'press a key' step is provided by another built-in function, **keyhit()**. This makes the program look for any key to be struck, and the keycode remains in a buffer. Using **keyhit()** in a **while** loop causes the program to hang up in a loop, waiting for a key to be hit. When you press any key, the program then continues, initialising the counter **n** once again, and opening the file for reading. Now it can happen that you don't have the correct disc in the drive when you are reading a file, and the next part of the program shows how to deal with this contingency. The pointer **fp** will be zero if no file exists, so that testing for (**fp==0**) allows you to print a message. In a real program, of course, you would want to return to the waiting step if the disc turned out to be the incorrect one, but in this example, the program simply stops if the **newchr** file is not on the disc. If the file is found, then a **while** loop reads it until the EOF character is found. The EOF has been defined at the start of the program as -1, the correct EOF for HiSoft C. We could, of course, have used -1 in place of EOF, but if you use **EOF** and **#define**, it's much easier to change a program so as to run on another machine (or another variety of 'C'). The **getc()** function is assigned to the integer **j** so that the EOF can be detected, and the conversion to characters is done simply by using **a[n++]=j**, in which the character is placed in the array of characters and the place

number incremented. When the loop ends because of the EOF character, the ‘\0’ is added to make the array into a true string. The file is closed, and the string of characters is printed. The last step looks mysterious. It reads the key that was struck at the **keyhit()** step, and assigns the value to **j**. The only reason for doing this is to avoid having a code in the keybuffer when the program ends. If there is a code in the buffer, then the prompt ‘Type y to run’ appears, and the code in the buffer, unless it is the code for ‘y’, causes a return to the editor. This is an important point to watch, because these stored key-codes can be a lot of stored trouble even when a program ends. Now you can start condensing the size of the program by merging actions as was illustrated previously.

String files.

The use of number and character files is seldom particularly important, except as parts of other files. That’s something that we shall take a close look at in the following chapter. For the moment, the important string file is one that we want to attend to. As you know, a string in ‘C’ is an array of characters which ends with a ‘\0’ marker. An array of strings can be dealt with in two ways. One is as an array that has two dimensions, such as **a[10][10]**, another is by keeping an array of pointers. Experienced ‘C’ programmers work as a matter of preference with pointers, and we have already had a taste of this when we saw that a pointer plus a subscript number could be used to refer to an item in an array. In the following example, we’ll make much more use of pointers by using an array of pointers to store a string array.

```
#define EOF -1
#define NULL 0
extern char *gets(),*fgets();
main()
{
  int n,*fp;
  char str[40],*sp;
  n=0;
  fp=fopen("strfil","w");
  do
  {
    sp=gets(str);
    fprintf(fp,"%s\n",str);
    n++;
  }
  while (n<=10);
  fclose(fp);
  printf("\nPress any key to read\n");
  while (!keyhit());
  fp=fopen("strfil","r");
  if (fp==0)
  {
    printf("\nNo such file");
  }
}
```

```

n=20;
while(sp=fgets(str,n,fp))
{
    if (sp=0) break;
    printf("%s",str);
}
fclose(fp);
n=rawin();
}
char *gets(s)
char *s;
{
    static int c;
    static char *cs;
    cs=s;
    while ((c=getchar())!=EOF && c!='\n') *c
s++=c;
*cs=0;
return((c==-1 && cs==s)?NULL:s);
}
char *fgets(s,n,fp)
char *s;
int n;
int *fp;
{
    static int c;
    static char *cs;
    cs=s;
    while (--n>0 && (c=getc(fp))!=EOF) if (
(*cs++=c)=='\n') break;
*cs='\0';
return((c==EOF && cs==s)?NULL:s);
}

```

Using pointers in a program to store a string array.

FIGURE 5.9

The program is illustrated in Figure 5.9. It's considerably longer and more complicated than any of the 'C' programs that we have looked at so far, and there are several new points embedded within it. The first point is that two routines from the library have been included in the program. So far, when we have included library programs, they have either been the type which returned an integer (or nothing), or they have been included by using **stdio.h** and **stdio.lib**. This time, two routines which return character pointers have been used, and placed following the main program. Because they are not simple integer returning routines, the compiler will need to be notified of them in advance. An alternative is to place the functions ahead of the main program. In this case, the 'forward declaration' is used, with

extern followed by a copy of the header for each function, **gets** and **fgets**. You will find the two functions in your **stdio.lib** file, and the forward declarations in **stdio.h**. The **gets()** function will get a string named **s** from the keyboard, and returns a string pointer which need not be used. In this example, the pointer has been assigned, but not used, and each expression could be simplified by omitting the assignment. The only reason for including it is to show that it can be done.

The first part of the program opens a file called 'strfil' which is intended to take a number of strings, counted by the **do..while** loop from 0 to 10, eleven in all. Each string is obtained from the keyboard by using **gets(str)**, a library function which assigns the string to the name that is supplied and returns a pointer to the first character. The string is then saved to a disc file by using the usual **fprintf** routine. Note that this has used **str** but it could just as easily have used **sp**. This is an important point, because it's easy to assume that you might need to use ***sp**, which would, in fact, give just the first character of the string. Once all of the strings have been read and filed, the file is closed, and the first part of the program ends. There is no attempt, in an example like this, to check that the number of characters in each string does not exceed a maximum, and if the dimensioning of **str** is exceeded, the program could crash. For a working program, some kind of protection against exceeding the limit would have to be included.

The replay starts with the 'Press any key' type of loop that we encountered earlier. The file pointer is allocated for a read file, and tested in case the file does not exist. Once again, no attempt is made to return to the waiting loop at this stage. The file reading loop makes use of the function **fgets**. This takes three parameters, **str,n,fp**, which denote the string, number of characters and file-pointer respectively. The function will read the file whose pointer is **fp** and return a string of **n-1** characters from the file. For a reason that we'll look at later, it looks as if it doesn't, but I can assure you that it does. Each string is then printed, and the printing line used **%s** to specify a string, but no **\n** to force a newline. This is because the strings already have newline characters when they are put into the file with **fprintf**.

So far, so good. For the moment we'll ignore the library functions, except to point out that they have to start with a declaration of type, and the asterisk which shows that they each return a pointer. The program will compile and run as you would expect, but when you test some of its characteristics, it seems to be misbehaving. In particular, when you enter a string of more than 20 characters, it appears to be replayed with no alteration. How is this possible, if the **fgets()** function is working? The penny drops when you count the number of strings. If you have put in one long string, you read it back as more than one string. It appears on the screen undivided because there was no newline character in it when you entered it. If you really want to see what the reading action does, then place a newline descriptor of **\n** in the **printf** line for the returned string. You'll see then that the long strings are read as strings of up to 19 characters each, this time separated by a newline; with the other strings separated by two newlines.

If, like me, you hate to have a mystery left unexplained, I'll deal with the **return** line of the library routines. The **test?action1:action2** line is a way of choosing to return one quantity or another. If the test is TRUE, then **action1** is taken, if the test is FALSE, then **action2** is followed. In the example of ***gets(s)**, the line is:

```
return((c==-1 && cs==s)?NULL:s)
```

so as to select which quantity to return. The reason for this line is that the function should return a pointer to NULL, address 0, if for any reasons a true pointer cannot be obtained. In any other case, the pointer should be **s**, which is the pointer obtained in the function. The test is **(c==-1 && cs==s)**, meaning that the character is the EOF character (EOF would have been safer here!) and the pointer to character position, **cs**, is still pointing where it was originally set, to **s**. If this is true, then nothing has been read into the string, and NULL is returned. For any other values, the pointer **s** is returned.

Chapter 6

More structured types.

We have come quite a long way in looking at examples and applications of 'C', but there are still plenty of topics to get to grips with. One of these is records, something that is not easy at the best of times, and more difficult if you have only ever programmed the Amstrad in BASIC. A **record** is a collection of items of data, which may all be of the same type or, more usually, of different types. What makes these items into parts of a **record** is that they are related. To take an example, suppose that you wanted to keep a record of membership of the local Golf Club. You would need the name and the address for each member. These would be strings, arrays of type **char**. You might also want to keep year of birth (because Juniors pay a reduced fee, and senior members pay only green fees), year of joining (members with ten or twenty years membership have special privilege years), and handicap. All of these last three items could also be stored as strings, because string entry and storage is easier. There might also be an entry for fees due (a real number in a real-life program) and whether paid or not to date.

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

```
#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
char *gets(s)
char *s;
```

```

{
static int c;
static char *cs;
cs=s;
while ((c=getchar())!=EOF && c!='\n')*c
s++=c;
*cs=0;
return s;
}
struct golf_club {
char name[20];
char address[40];
char birth[5];
char join[5];
char hcap[3];
int subs;
char paid;
}G;
main()
{
int j;
char c,s[5];
for (j=1;j<=1;j++)
{
rawout(12); /*clear screen*/
printf(N);
gets(G.name);
printf(A);
gets(G.address);
printf(YB);
gets(G.birth);
printf(YJ);
gets(G.join);
printf(H);
gets(G.hcap);
printf(S);
gets(s);
G.subs=atoi(s);
printf(P);
c=getchar();
tolower(c);
G.paid=c;
printf("\nPress any key...\n");
j=rawin();
printf(N);
printf("%s\n",G.name);
printf(A);
printf("%s\n",G.address);

```

```

printf(YB);
printf("%s\n",G.birth);
printf(YJ);
printf("%s\n",G.join);
printf(H);
printf("%s\n",G.hcap);
printf("Subscription:- ");
printf("#%d.%d\n", (G.subs)/100, (G.subs)
%100);
printf(F);
printf(" %c\n",G.paid);
}
}
atoi(s)
char *s;
{
    static int c,value,sign;
    while(isspace(*s))++s;
    value=0;
    sign=1;
    if (*s=='-')
    {
        ++s;
        sign=-1;
    }
    else if (*s=='+') ++s;
    while (isdigit(c=*s++))value=10*value+c
-'0';
    return sign*value;
}

```

The arrangement of a structure, showing how it is declared and used.

FIGURE 6.1

We'll start by considering what we need to do in Figure 6.1 to declare a **structure**, using as an example the Golf Club illustration above. The first action, as usual, is to declare any constants. In this illustration, we shall make the total membership of the club a constant, equal to 2, because this is just an example. By making this a small number, you can see how the program works without wearing out your typing finger(s). The program which is illustrated is a long one, and the intention is that you type it once and save it. The developments can then be added by editing the 'starter' program, so that you'll find a few items in the starter which aren't needed immediately. The **#define** lines define a number of useful items, along with a set of messages which will be used in the entry and reprinting sections of the program. The function **char *gets()** is then defined, so as to avoid the business of declaring it as an **extern**, as we used it previously.

The important part, however, is what follows in the **struct** declaration. The name of the **record** is given as **golf__club**. This is a reminder only, because though we could use this as a variable name, it's rather unwieldy, as you'll see. The name that is used here is sometimes called the 'tag' of the structure. The structure **golf__club** is declared, and what follows within curly brackets must be a list of the **fields** of the record, meaning the items that make up the record. I have typed these indented, with one item per line, to make them more obvious. Like any other declaration, the items could be grouped with all the **char** names following the **char** heading, separated by commas. The name and address fields are both strings, but with different numbers of characters. The two years and the handicap are also taken as strings, with the dimensioning for five characters in the year because there will be four digits and the '0' which marks the end of the string. If you don't dimension adequately, the program will compile and run, but the results will be decidedly odd! The subscription amount should be a 'float' number— in a program which was seriously intended to keep records of this type, the amount of the subscription would be calculated from a formula and printed when required, but in this example, I have made it an entered integer item. The letter **paid** is of type **char**, and will be used for a 'Y' or 'N' reply, because the subscription will either be paid or not— this club doesn't allow instalment payments! The end of the definition of the fields of this record is marked with the usual } sign. All of this definition occurs before the start of the main program, and following the curly bracket which ends the structure definition, we must have a semicolon. If there is only a semicolon, then the name of the structure can be assigned later by using a line like:

```
golf__club G;
```

By using the syntax: }**G**; , however, we can use **G** to mean a structure of type **golf__club** without using another line, which is much more convenient. We could, if we liked, declare other names in this way, such as: }**G,H,J**; so as to mean that **G,H,K** were all names for structures of the type **golf__club**.

The main program then starts, with an integer declared for a counting loop, a character for the 'y' or 'n' entry, and a short string for getting the subscription amount. Apart from **scanf**, the input functions of 'C' are biased to character or string entry, so it makes sense to use string entry for almost everything since the **gets()** function is being used in any case. There is a function, **atoi** which will convert a string of digits into an integer. The program then starts with a loop. It's a once only loop in this example, because there's no point in using the loop more than once until the program has been expanded a bit, but I've put it in to show how this would be done. In the first section, the screen is cleared, and information is prompted for and entered. The prompt messages are obtained by using the string constants **N, A, YB...** which were defined at the start of the program. The important point to note is how the inputs are assigned. For the name entry, for example, we use **gets(G.name)**. This calls function **gets()**, and assigns the string that it gets from the keyboard to variable **G.name**. This is the way that we can select one item (or field) of a record, using the structure variable name, then a full-stop, then the item name. This syntax lets you assign to an item in a structure or print an item. Later on, we'll see that if you want to do anything more complicated you need, as always, to use pointers.

The rest of the information is then entered in the same way, with the subscription being requested in pence. Now this is just a convenience, and it would be just as easy, since a string is being entered, to enter in conventional pound-dot-pence or pound-dash-pence form. The parts could then be separated and converted to separate integers to avoid the problems of not having float numbers. This example is supposed to be a simple one, however, so we'll stick to simple ways. The number that is entered is converted to an integer by function `atoi` and stored as `G.subs`. If your subscription is too large to store as an integer you're spending too much on golf! The last item is a 'y' or 'n' entry of a single character, and the function `tolower` is used in case the (SHIFT) key was used. The data is then 'replayed', simply to show that it all works. The titles are printed using, mainly, the string constants, and the answers are obtained from the structure name and portions. Note that the subscription is printed in pounds and pence by making use of integer division and modulus operators. The use of `j=rawin()` for the 'Press any key' stage avoids difficulties with stored characters, and allows the program to end with the usual 'Type y to run:' notice.

Filing structures.

The structure in 'C' is so useful as a way of packing information into groups that we need some way of recording structures on disc. It would be pleasant if we had a structure filing statement which allowed a complete structure to be put on to disc simply by using the structure name. This, however, can't be done, and we have to record the items of a structure one by one. Though this could be done as part of a main program, we'll learn a lot more about the use of structures and pointers if we make the structure filing routine part of a function. The important point here is that you can't pass the name of a structure to a function and expect it to do anything about it. You can, however, pass a pointer to a structure by using the `&` sign with the structure name. This is needed so often that 'C' has a special way of indicating the items in a structure by way of the pointer. For example, if `sp` is the pointer to a structure, then `sp->item` will refer to the item in the structure. The `->` sign uses the minus and greater-than signs together.

```
#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
#define W "\nPrepare disc for data\n"
#define M "\nPress any key to proceed.\n"
"
#define MM "\nPress @ key to end, any ot
```

```

her to continue entry.\n"
char *gets(s)
char *s;
{
    static int c;
    static char *cs;
    cs=s;
    while ((c=getchar())!=EOF && c!='\n')*c
s++=c;
    *cs=0;
    return s;
}
struct golf_club {
    char name[20];
    char address[40];
    char birth[5];
    char join[5];
    char hcap[3];
    char subs[6];
    char paid[2];
}G;
main()
{
int j,count,*fp;
count=0;
printf(W);
printf(M);
j=rawin();
fp=fopen("clubdat","w");
do
{
count++;
rawout(12); /*clear screen*/
printf(N);
gets(G.name);
printf(A);
gets(G.address);
printf(YB);
gets(G.birth);
printf(YJ);
gets(G.join);
printf(H);
gets(G.hcap);
printf(S);
gets(G.subs);
printf(P);
gets(G.paid);
filit(fp,&G);
}
}

```



```

printf(MM);
j=rawin();
}
while (j!='@');
fclose(fp);
fp=fopen("dim","w");
fprintf(fp,"%d",count);
fclose(fp);
}

filit(filp,sp)
struct golf_club *sp;
int *filp;
{
fprintf(filp,"%s\n%s\n",sp->name,sp->ad
dress);
fprintf(filp,"%s\n%s\n%s\n",sp->birth,s
p->join,sp->hcap);
fprintf(filp,"%s\n%s\n",sp->subs,sp->pa
id);
}

```

Entering data into a structure and recording it on disc.

FIGURE 6.2

A sample structure-filing program is illustrated in Figure 6.2. This has been constructed out of part of the program of Figure 6.1, omitting the printing sections. Three new messages have been added, though one needs tidying up with an additional `\n` to force a newline and prevent a word from being split. A disc file is opened, with the name "clubdat", for writing. The setup of the structure is the same as before, as is the entry of information. After the information on payment of subscriptions has been entered, a function `filit()` is called to place the data of the structure on file. This function uses the file pointer for the disc, `fp`, and also the pointer address `&G` for the structure. The program then hangs up waiting for a reply key to be pressed. If this is the `@` key, then the program closes files and terminates. Any other key will continue the `do..while` loop. I found that in this type of loop, the `j=rawin()` step left something in the buffer which would be taken as a string by `gets()`. This caused the next cycle of inputs to disregard the 'Name' step, and was cured this time by adding another `gets()` step following `j=rawin()`. Note that this was on a CPC464, and it's possible that later Amstrad machines might not exhibit this problem.

The meat of the program therefore lies in the function `filit`, which uses two parameters. One parameter is the filepointer, `filp`, an integer, and the other is the structure pointer `sp` which is declared as `struct golf_club *sp`. This declaration is that `sp` is a pointer to a structure of type `golf_club`. The fields of the structure are then sent to the file, using `fprintf` statements. The name and address strings are sent first, using "`%s%s`" as the specifier for the two strings, and with `sp->name`, `sp->address` as the separate fields. The other fields of the structure are dealt with in the same way, remembering that `sp->subs` is an integer, and `sp->paid` is a single character. Since the items that are to be recorded are stored in a buffer until the buffer fills or the file is closed, you don't necessarily hear much activity from the disc at the time when this function runs.

Reading back structures.

Reading back a file of structures from the disc normally uses **fscanf**, but you must remember that this function works with pointers. In addition, **fscanf** will take the end of a string as being the first 'white space' in the string, meaning the first blank or any other character which does not 'belong' in a string, such as the (TAB) key or the (space) key. This makes **fscanf** more suited for files of integers, or of strings which can be guaranteed to have no spaces in them, but it's not very useful for the type of string that we now want to read, with names and addresses. Fortunately, the library contains the useful **fgets()** function, which is very similar to **gets()**, but with subtle differences. This function can be used to read back all of the recorded strings, and will not give trouble if any 'white space' is found in a string.

That doesn't mean that everything is plain sailing, because when you use a library function you have to read the small-print (or its listing) to see just what it will do with the data. The similarity between **fgets** and **gets** is close, but one difference is *very* important. Whereas **gets** will read a string of characters, including the (RETURN)/(ENTER) character, and then replace the (RETURN)/(ENTER) character by a zero to act as string terminator, **fgets** does not do this. The **fgets** function reads a string until the '**\n**' character is found, and then adds a zero to the end of this. This makes the string longer. For example, if year of birth is entered as a string of four characters, it will be recorded as five characters (the '**\n**' being the fifth), and will be returned into the program as a string of six characters in all, including the '**\n**' and the '**\0**'. This means that we have to be careful about dimensioning the strings that we shall read into, because it's easy to fall into the trap of assuming that the string we read back will be the same as we recorded. This behaviour of **fgets** can be changed by decrementing **cs** in the **fgets** function before it is equated to zero, but in this example, I have used the library function simplified but not returning anything different. The other point to watch is that **fgets** takes three parameters, the string name, a string length number, and the filepointer. The string-length number decides how many of the characters of the string are read, and the function reads characters until this number is exceeded or until a newline character is found. Unless you are *very* sure of your string lengths, it's better to provide generous values of length, so that the newline character ends the reading action. If, of course, all of the strings were tested for length before recording, there's no objection to counting them back precisely. What you need to remember, however, is that the number that you provide for string length in **fgets** must be the complete string length, including the ending zero or newline character.

With these warnings in mind, we can now look at what is needed to read back the string file that was created by the program of Figure 6.2.

```
#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
```

```

#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
char *fgets(s,n,fp)
char *s;
int n;
int *fp;
{
    static int c;
    static char *cs;
    cs=s;
    while(--n>0 && (c=getc(fp))!=EOF) if ((
*cs++=c)=='\n') break;
    *cs='\0';
    return s;
}
struct golf_club {
    char name[20];
    char address[40];
    char birth[6];
    char join[6];
    char hcap[4];
    char subs[6];
    char paid[3];
}G[50];
main()
{
    int x,j,count,*fp;
    char c;
    do
    {
        fp=fopen("dim","r");
    }
    while(!testit(fp));
    fscanf(fp,"%d",&count);
    fclose(fp);
    fp=fopen("clubdat","r");
    if (count>5) count=5;
    for (j=0;j<=count-1;j++)
    {
        fgets(G[j].name,20,fp);
        fgets(G[j].address,40,fp);
        fgets(G[j].birth,6,fp);
        fgets(G[j].join,6,fp);
        fgets(G[j].hcap,5,fp);
        fgets(G[j].subs,7,fp);
        fgets(G[j].paid,3,fp);
    }
}

```

```

fclose(fp);
printf("\nPress any key...\n");
j=rawin();
rawout(12);
for (j=0; j<=count-1; j++)
{
printf(N);
printf("%s\n",G[j].name);
printf(A);
printf("%s\n",G[j].address);
printf(YB);
printf("%s\n",G[j].birth);
printf(YJ);
printf("%s\n",G[j].join);
printf(H);
printf("%s\n",G[j].hcap);
printf("Subscription:- ");
x=atoi(G[j].subs);
printf("#%d.%d\n",x/100,x%100);
printf(P);
printf("%s\n",G[j].paid);
}
}
atoi(s)
char *s;
{
static int c,value,sign;
while(isspace(*s))++s;
value=0;
sign=1;
if (*s=='-')
{
++s;
sign=-1;
}
else if (*s=='+') ++s;
while (isdigit(c=*s++))value=10*value+c
-'0';
return sign*value;
}
testit(k)
int k;
{
if(k!=0) return k;
else
{
printf("\nNo such file.");
}
}

```

```

printf("\nFind correct disc and");
printf("\npress any key to try again.")
;
}
}

```

Reading and printing a file of structures.

FIGURE 6.3

The reading program is shown in Figure 6.3, and it starts in the usual way with the **#define** steps, the **fgets** function, and the structure declaration. Notice that this time the structure has been named as **G[50]**, an array of structures. This allows us to store the structures as an array, and demonstrate how an array of structures is handled. When the structures were recorded, a count number was included in a separate file, **dim**. This is one useful way of ensuring that the writing and the reading programs do not get out of step. In BASIC, this number could be read and then used to dimension the array. In 'C' this type of thing is not so easy, because the structure has to be dimensioned before the main program starts. If the declaration is made before the start of the main program (that is, all of **struct golf_club** but without the **G[50]**), then the declaration of the name (**G**) and the dimensioning could be done in a function, with the structure not used in the main program. This function would have to be called after the count number was loaded from disc. In this example, however, we've kept to simpler methods, and the count number is used simply to operate the playback loop. Once again, in a real program, the count number would be compared with the dimensioning of the array to check that no attempt was made to overfill the array. In this example, the line:

```
if (count>5) count=5
```

is deliberately put in to keep the amount of data down, though the dimensioning allows for 50 structures to be used. The use of **count** also allows a test for the correct disc being in place before the main data is read. All of this can be replaced, and the count ignored, if the end of file is used as a means of detecting the end of the structures on the disc, but this is not particularly easy to do.

The example in Figure 6.3 uses the count number to set up a loop which will read in structures. The loop runs from 0 to count-1, rather than from 1 to count, so that the first structure (**G[0]**) is not wasted. If a structure name by itself, such as **G[0]**, is used in a function it has to be represented by its pointer, but when parts of a structure are being read, as in **G[j].name**, these can be used directly. The **fgets** lines read in all of the parts of each structure, after which the file is closed. The data is then displayed, in this case simply to confirm that it has been correctly read and placed in the array. The only novelty in the printout loop is the printing of the subscription amount. This is done using function **atoi** which converts a string into an integer, providing that the string consists of digits. The integer **x** is obtained from the **subs** string, and this is printed in pound, pence fashion by making use of integer division and modulus operators.

Working with structure files.

The example of Figure 6.3 showed the reading part of a file being followed by a simple printout of each record. A much more normal action would be to pick out one record, or to sort the records into alphabetical order. Now picking out one record is fairly simple, as the amended program of Figure 6.4 shows.

```
#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
char *fgets(s,n,fp)
char *s;
int n;
int *fp;
{
    static int c;
    static char *cs;
    cs=s;
    while(--n>0 && (c=getc(fp))!=EOF)
    {
        if (c=='\377')continue;
        if ((*cs++=c)=='\n') break;
    }
    *cs='\0';
    return s;
}
char *getstr(s)
char *s;
{
    static int c;
    static char *cs;
    cs=s;
    while((c=getchar())!=EOF )
    if ((*cs++=c)=='\n')break;
    *cs=0;
    return s;
}
```

```

struct golf_club {
    char name[20];
    char address[40];
    char birth[6];
    char join[6];
    char hcap[4];
    char subs[6];
    char paid[3];
}G[50];
main()
{
    int x,j,count,*fp;
    char c,s[20];
    /*Length not checked*/
    do
    {
        fp=fopen("dim","r");
    }
    while(!testit(fp));
    fscanf(fp,"%d",&count);
    fclose(fp);
    fp=fopen("clubdat","r");
    if (count>5) count=5;
    for (j=0;j<=count-1;j++)
    {
        fgets(G[j].name,20,fp);
        fgets(G[j].address,40,fp);
        fgets(G[j].birth,6,fp);
        fgets(G[j].join,6,fp);
        fgets(G[j].hcap,5,fp);
        fgets(G[j].subs,7,fp);
        fgets(G[j].paid,3,fp);
    }
    fclose(fp);
    rawout(12);
    printf("\nPlease type name required.");
    getstr(s);
    x=false;
    for (j=0;j<=count-1;j++)
    {
        if (strcmp(s,G[j].name))continue;
        else;
        {
            printf(N);
            printf("%s\n",G[j].name);
            printf(A);
            printf("%s\n",G[j].address);
            printf(YB);
        }
    }
}

```

```

printf("%s\n",G[j].birth);
printf(YJ);
printf("%s\n",G[j].join);
printf(H);
printf("%s\n",G[j].hcap);
printf("Subscription:- ");
x=atoi(G[j].subs);
printf("#%d.%d\n",x/100,x%100);
printf(P);
printf("%s\n",G[j].paid);
x=true;
}
break;
}
if (x==false)printf("\nName not found.")
;
}
atoi(s)
char *s;
{
static int c,value,sign;
while(isspace(*s))++s;
value=0;
sign=1;
if (*s=='-')
{
++s;
sign=-1;
}
else if (*s=='+') ++s;
while (isdigit(c=*s++))value=10*value+c
-'0';
return sign*value;
}
testit(k)
int k;
{
if(k!=0) return k;
else
{
printf("\nNo such file.");
printf("\nFind correct disc and");
printf("\npress any key to try again.")
;
}
}
strcmp(s,t)
char *s,*t;

```



```

{
  while (*s==*t)
  {
    if (!*s) return 0;
    ++s;
    ++t;
  }
  return *s-*t;
}

```

Reading structures into an array so that they could be sorted.

FIGURE 6.4

The principle is once more to read in the records as an array of structures, using **fgets()**, which will result in each string ending with a ‘\n’ and a zero. When the array has been read in, the program prompts for the name that is being searched for. This is entered, using a function called **strget(s)**. The reason for not using **gets(s)** is that **gets(s)** will always replace the ‘\n’ character by a ‘\0’, and to match the string that we are entering we need it to contain both of these characters. The function **strncmp** is therefore modelled after **fgets()** rather than after **gets()**, in that it has a parameter that tells it how many characters to copy. In addition, this program introduces the idea of comparing strings. This cannot be done in the familiar BASIC IF A\$=B\$. way. The name of a string in ‘C’ is simply a pointer to its first character, and you can’t expect the pointers to two different strings to be equal. Any comparison such as:

```
if (s==G[j].name)....
```

is doomed to failure. To compare strings, you need to use a string comparison function, such as **strcmp**. The two strings are passed as parameters to this function, and it will return zero (false!) if the strings are equal, true if not. In fact, if the strings are not equal, the number that is returned is an integer equal to the difference between the string pointer numbers, but anything which is not zero is counted as true for the purposes of a test.

The test for equality of strings is used to make the loop run faster by causing a **continue** in the **for** loop if the strings are not equal. When a matching string is found, the **else** section runs, printing the details for the selected name. Before the loop started integer **x** was made equal to ‘false’ (zero), and if a string match is found, this integer **x** becomes true and the loop breaks. In this way, the loop runs fast until a matching string is found, and then breaks immediately afterwards. The integer **x** is used after the loop ends to print a suitable message if no matching name has been found. It works perfectly – unless you want to get the first name on the file! The problem with this one is that ‘C’ reads the ASCII file starting one character earlier than it ought to. This usually means that the first character of the file comes in as the character that corresponds to (CTRL-Z) in ‘C’, 255, and appears as a double-headed horizontal arrow when printed. You can’t enter this character from the keyboard when you are asking for a name, so that the only thing to do is to remove it either from the file or from the readback. This has been done in the program of Figure 6.4 by modifying the **fgets()** routine. In the modified version, the character is tested for equality

to 255, and the while loop continues if this is true. The curious appearance of the test is due to the way that a number like 255 has to be expressed. Tests of this type in 'C' use octal, scale of eight code, so that 255 denary comes out as 377 octal. There is more information on octal code in the Appendix. With this test in place, the 255 character is rejected. If the file contains this character, of course, this would cause trouble, but an ASCII coded file would never contain this character and if you are working with integers then you'll use `fscanf()` rather than `fgets` for file reading. The CP/M version of 'C' behaves much better in this respect. You might like to work on this program now so that the selection of a name routine will loop until a blank name is entered (by pressing (ENTER) without typing a name).

Sorting a file.

In BASIC, there is nothing that corresponds to the structure. This makes actions such as sorting very tedious, because each field of a record has to be represented by an array item or as part of a string. Sorting is never easy, but in HiSoft C you do at least have the advantage of a sort routine in the library.

```

#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
char *fgets(s,n,fp)
char *s;
int n;
int *fp;
{
    static int c;
    static char *cs;
    cs=s;
    while(--n>0 && (c=getc(fp))!=EOF)
    {
        if (c=='\377')continue;
        if ((*cs++=c)=='\n') break;
    }
    *cs='\0';
    return s;
}

```

```

struct golf_club {
    char name[20];
    char address[40];
    char birth[6];
    char join[6];
    char hcap[4];
    char subs[6];
    char paid[3];
}G[5],*Gp;
main()
{
int y,i,it,z,x,j,count,*fp;
/*Length not checked*/
do
{
fp=fopen("dim","r");
}
while(!testit(fp));
fscanf(fp,"%d",&count);
fclose(fp);
fp=fopen("clubdat","r");
if (count>5) count=5;
for (j=1;j<=count;j++)
{
fgets(G[j].name,20,fp);
fgets(G[j].address,40,fp);
fgets(G[j].birth,6,fp);
fgets(G[j].join,6,fp);
fgets(G[j].hcap,5,fp);
fgets(G[j].subs,7,fp);
fgets(G[j].paid,3,fp);
}
fclose(fp);
rawout(12);
printf("\nPress any key for list");
Gp=G;
y=1;
while (y<count)
y=2*y;
do
{
y=(y-1)/2;
it=count-y;
for (i=1;i<=it;i++)
{
j=i;
do

```

```

        {
            z=j+y;
            if (strcmp(G[z].name,G[j].name)<=0)
            {
                swap (Gp+z,Gp+j,sizeof (struct golf_c
1ub));
                j=j-y;
            }
            else j=0;
        }
        while (j>0);
    }
} while (y!=1);
for (j=1;j<=count;j++)
{
    x=rawin();
    printf(N);
    printf("%s\n",G[j].name);
    printf(A);
    printf("%s\n",G[j].address);
    printf(YB);
    printf("%s\n",G[j].birth);
    printf(YJ);
    printf("%s\n",G[j].join);
    printf(H);
    printf("%s\n",G[j].hcap);
    printf("Subscription:- ");
    x=atoi(G[j].subs);
    printf("#%d.%d\n",x/100,x%100);
    printf(P);
    printf("%s\n",G[j].paid);
    printf("\nPress any key...");
}
}
atoi(s)
char *s;
{
    static int c,value,sign;
    while(isspace(*s))++s;
    value=0;
    sign=1;
    if (*s=='-')
    {
        ++s;
        sign=-1;
    }
    else if (*s=='+') ++s;

```

```

    while (isdigit(c=*s++))value=10*value+c
    -'0';
    return sign*value;
}
testit(k)
int k;
{
if(k!=0) return k;
else
{
printf("\nNo such file.");
printf("\nFind correct disc and");
printf("\npress any key to try again.");
;
}
}
strcmp(s,t)
char *s,*t;
{
while (*s==*t)
{
if (!*s) return 0;
++s;
++t;
}
return *s-*t;
}

```

A structure array program which incorporates a sort routine.

FIGURE 6.5

The illustration of Figure 6.5 shows a Shell-Metzner type of sort routine used to sort a list of records in alphabetical order of names as typed in the file. Now this is not the Shell-Metzner sort which is included in the library, because that one is a rather complicated general-purpose one. Nor is it the sort that is mentioned in the manual, from Kernighan & Ritchie's book, because it won't compile into HiSoft C as it stands. The problem is that there are three loops, and this causes a 'Too many operators' message when you try it. The one in the library only just fits in. Because of all that, and for the sake of variety also, I have put in another Shell type of sort, adapted from a version which I wrote in Pascal.

The program reads the count number, and then reads the structures into an array. This time, simply because it's more convenient for the sort routine, the array numbers start with 1 rather than with zero, but that's the only change up to the point where the message 'Press any key for list' is printed. After that, things get more complicated. In the structure declaration, the usual **G[5]** has been supplemented by ***Gp**, making **Gp** a pointer to a structure. Simply declaring that **Gp** is a pointer, however, doesn't make it point to

anything, and the statement **Gp=G** is needed to make **Gp** a pointer to the start of structure **G**. This is a very important step, and omitting it is one of the most common errors in the use of pointers. If your pointer hasn't been set to point at something, then trouble, in the shape of a major program crash, can't be far behind. Why do we need the pointer anyhow? The answer is that in the sort routine we shall want to change the pointers to different members of the array. Changing pointers involves exchanging only two numbers, rather than the set of strings (or whatever else is used) in a structure. For this reason, it's fast and simple.

I won't go into details of how the Shell-Metzner sort works, because it's a standard routine that you can find described at length in many other books. The important features of the sort (starting at the statement **y=y+1**) are the test and exchange steps. The test uses **strcmp** as you would expect, with **G[].name** being used as a basis of comparison. The exchange step uses the built-in function **swap**, and the parameters that are supplied make use of the string pointers, along with the **sizeof** statement. The **sizeof** statement, as its name suggests, will provide an integer, the number of bytes of memory allocated to a variable. In this case, it's the structure **golf_club** whose size we want to pass on. We can count it up for ourselves by adding the bytes allocated to each part of **golf_club**, and it comes to 85, but by using **sizeof** the counting is automatic, so that if you alter the structure dimensioning you don't have to alter the swap routine. The swap routine exchanges pointers, if need be, and when the sort is completed, the structures will be in order of names. The important point is that, because of the **Gp=G** step, you can print out this new order using **G[j]**, you don't have to use pointer **Gp** unless you want to. This is the value of altering pointers in this way, because the pointers can be altered in a subroutine and the alteration will affect the result of a printout in the main routine. In this example, the whole sort routine is, unusually, in the main program, simply to avoid the problems of passing parameters until you have seen an example of the straightforward version. The remaining parts of the program are straightforward, and a 'Press any key' step has been put into the loop which prints the details so that you don't lose data because of scrolling.

The next problem is how you would alter this program so as to sort in order of ascending handicap numbers. I have not used handicap numbers in the strict golfing sense here, so please don't write to say that the range should be from about +4 to -36! It's not so easy as you might think, because simply using **G[].hcap** does *not* do what you might think. This change will work if the handicaps are single figures (OK for an Open championship, perhaps), but not for double figures. The reason is that a number like 20 is taken as being less than the number 6. This is because the **G[].hcap** field is a string, not an integer, and the comparison goes one character at a time. When a '2' is found as the first character, this is taken as being less than the '6', and so the second character of the '20' is never considered. How do we get round that one?

The answer is a simple alteration to the comparison line, using **atoi** in place of **strcmp**. The line now becomes:

```
if (atoi(G[z].hcap)<=atoi(G[j].hcap))
```

In this line, if the handicap of member **z** is less than or equal to the handicap of member **j**, then the swap will be carried out as before. The important point, once again, is that the effect of swapping an **entire record** can be carried out by swapping pointers.

Now suppose that we wanted to sort the list of names in some other way. In Figure 6.5, we chose to sort in alphabetical order name, and we have seen also how to sort by ascending handicap number. Sorting by **descending** handicap is trivial, all you need to do is to reverse the \leq sign in the comparison line into a \geq sign. The important change is to be able to sort 'by another field'. In plain language, this means sorting by surname order, or forename order, by age, or by any other feature. This is easy enough if you want the sort to be permanent. If, for example, you **always** wanted the program of Figure 6.5 to sort the records in order of handicap, then all you have to do is to carry out the changes that we have looked at. It's not quite so straightforward when sorting might be wanted by any of a number of fields, selected when the program runs. The obvious way to do this is to introduce a menu stage in the main program, and select each time the comparison that is needed when the program of Figure 6.5 runs.

```

#define EOF -1
#define NULL 0
#define total 2
#define true 1
#define false 0
#define N "Name - "
#define A "Address - "
#define YB "Year of birth - "
#define YJ "Year of joining - "
#define H "Handicap - "
#define S "Subscription in pence - "
#define P "Paid, Y or N - "
char *fgets(s,n,fp)
char *s;
int n;
int *fp;
{
    static int c;
    static char *cs;
    cs=s;
    while(--n>0 && (c=getc(fp))!=EOF)
    {
        if (c=='\377')continue;
        if ((*cs++=c)=='\n') break;
    }
    *cs='\0';
    return s;
}
struct golf_club {
    char name[20];
    char address[40];
    char birth[6];
    char join[6];
    char hcap[4];
    char subs[6];
}

```

```

    char paid[3];
    }G[5],*Gp;
main()
{
int s,y,i,it,z,x,j,count,*fp;
char c;
/*Length not checked*/
do
{
fp=fopen("dim","r");
}
while(!testit(fp));
fscanf(fp,"%d",&count);
fclose(fp);
fp=fopen("clubdat","r");
if (count>5) count=5;
for (j=1;j<=count;j++)
{
fgets(G[j].name,20,fp);
fgets(G[j].address,40,fp);
fgets(G[j].birth,6,fp);
fgets(G[j].join,6,fp);
fgets(G[j].hcap,5,fp);
fgets(G[j].subs,7,fp);
fgets(G[j].paid,3,fp);
}
fclose(fp);
do{
do{
rawout(12);
printf("\nPlease select sort method.");
printf("\n 1. Name.");
printf("\n 2. Age.");
printf("\n 3. Length of membership.");
printf("\n 4. Handicap.");
c=getchar();
c-=48;
}while(c<0 || c>4);
Gp=G;
y=1;
while (y<count)
y=2*y;
do
{
y=(y-1)/2;
it=count-y;
for (i=1;i<=it;i++)

```



```

{
    j=i;
    do
    {
        z=j+y;
        s=false;
        switch(c){
        case 1:if (strcmp(G[z].name,G[j].name
)<=0) s=true;break;
        case 2:if (atoi(G[z].birth)<=atoi(G[j
].birth))s=true;break;
        case 3:if (atoi(G[z].join)<=atoi(G[j]
.join))s=true;break;
        case 4:if (atoi(G[z].hcap)<=atoi(G[j]
.hcap))s=true;break;
        }
        if (s) {
            swap (Gp+z,Gp+j,sizeof (struct golf_c
lub));
            j=j-y;
        }
        else j=0;
    }
    while (j>0);
}
} while (y!=1);
printf("\nPress any key...\n");
for (j=1;j<=count;j++)
{
    x=rawin();
    printf(N);
    printf("%s\n",G[j].name);
    printf(A);
    printf("%s\n",G[j].address);
    printf(YB);
    printf("%s\n",G[j].birth);
    printf(YJ);
    printf("%s\n",G[j].join);
    printf(H);
    printf("%s\n",G[j].hcap);
    printf("Subscription:- ");
    x=atoi(G[j].subs);
    printf("#%d.%d\n",x/100,x%100);
    printf(P);
    printf("%s\n",G[j].paid);
    printf("\nPress any key...");
}

```

```

    printf("\n@ Key terminates display");
}while (x=rawin()!='@');
}
atoi(s)
char *s;
{
    static int c,value,sign;
    while(isspace(*s))++s;
    value=0;
    sign=1;
    if (*s=='-')
    {
        ++s;
        sign=-1;
    }
    else if (*s=='+') ++s;
    while (isdigit(c=*s++))value=10*value+c
    -'0';
    return sign*value;
}
testit(k)
int k;
{
    if(k!=0) return k;
    else
    {
        printf("\nNo such file.");
        printf("\nFind correct disc and");
        printf("\npress any key to try again.")
    }
}
strcmp(s,t)
char *s,*t;
{
    while (*s==*t)
    {
        if (!*s) return 0;
        ++s;
        ++t;
    }
    return *s-*t;
}

```

Amending the program so as to sort by any field.

FIGURE 6.6

Figure 6.6 shows the amendments that can be made to allow a choice of four possible sort fields, in this case, surname, year of birth, joining year or handicap. The sort and display part of the program has been arranged so that it will loop until the @ key is pressed. This allows you to enter data from the disc file and then test the action of the **switch** function.

Now look at the program of Figure 6.6 in detail. Two more variables have been declared. One is **c** which will be used to accept a letter choice when you are asked to decide which field to use for sorting. The other is **s**, which is an integer variable that will be used to decide whether to swap two records or not.

The selection action starts immediately after the data has been read in from the disc. The choices are listed, and you are asked to select by number. This is done within a **do** loop, so that when the number that you use is tested the menu will repeat until a correct choice is made. This, incidentally, causes the menu to perform several flashes at times when several characters are in the keyboard buffer. When an acceptable number is pressed, the function returns to the main program with the value of a number assigned to **c**. This is then used in the sorting function. In the record exchanging part of this routine, variable **s** is set to false and the lines that follow **switch** carry out the selection of field. In these lines, then, the letter which has been coded as **c** is used to select the correct comparing action, testing the correct field of each record. As a result of that test, **s** will be either TRUE or FALSE. If **s** is TRUE, then the test in the exchange portion of the sort routine will cause the records to be exchanged. Note that you don't have to type:

```
    if (s==true)....  
or  
    if (s!=0)....
```

only **if (s)...**, because this is the correct syntax in 'C' as it is in many versions of BASIC. The rest of the program then proceeds in the usual way.

This method of choosing which field to sort by can be used when the number of records is comparatively small, but causes problems when a large number of records are used. The problem is one of sorting time. The **switch** selection line has been placed in a part of the sorting loop which is repeated very many times during a sort, and as a result, has a large effect on the total time that is taken. Because 'C' is a compiled language, the effect is not so serious as it would be in BASIC, but it can make the sort action irritatingly slow. If you need to exercise a choice of sort field for a list of records which consists of a large number of items, all held in the memory, then a faster option is to have as many sort functions as you want sort fields, and to select the complete function. You might, for example, have functions **sortname**, **sortncap** and **sortage** which were selected by the value of **c**. Each of these functions would be identical apart from the test line in each one. This method requires more code, but runs much faster. The reason is that the choice is not having to be enforced each time two items are being compared. The choice is made once and used to select a function which can then run unencumbered. As usual, when you design a program you can design it to run fast, or you can design it to be comparatively short, but you can't normally have both! The alternative is to write a sort routine to which the parameters can be passed, but for a structure this is extremely difficult because you have to be able to pass the field names such as **G[j].name**, not just the array name.

Record nests and choices.

So far, each record that we have illustrated has consisted of items that are simple variables. We can, however, use records which consist partly or completely of **other records!** Structures which are a part of another structure are called 'nested' structures, and typically they are used to hold details of an entry. We might, for example, have an entry called **birth** which would require the details of day, month and year of birth. This could be provided by making **birth** a structure in itself, with day, month and year items of that structure. Figure 6.7 shows how this provision for nested structures can be used.

```
#define max 2
char *getstr(str,count)
char *str;
int count;
{
    static int c;
    static char *cs;
    cs=str;
    while(--count>0 && ((c=getchar())!='\n'
))
        *cs++=c;
    *cs=0;
    return str;
}
struct name{
    char sur[20];
    char frn[20];
};
struct dob{
    int day;
    int month;
    int year;
};
struct person{
    struct name memnam;
    struct dob birth;
    char phone[16];
}member[max+1];
main()
{
    char reply[18];
    int j;
    for (j=1; j<=max; j++)
    {
        printf("\nSurname - ");
        getstr(member[j].memnam.sur,20);
        printf("\nForename - ");
```

```

    getstr(member[j].memnam.frn,20);
    printf("\nDay of birth 1-31 -");
    getstr(reply,4);
    member[j].birth.day=atoi(reply);
    printf("\nMonth of birth 1-12 -");
    getstr(reply,4);
    member[j].birth.month=atoi(reply);
    printf("\nYear of birth - ");
    getstr(reply,6);
    member[j].birth.year=atoi(reply);
    printf("\nPhone number - ");
    getstr(member[j].phone,17);
}
rawout(12);
printf("\n%20s", "LIST");
printf("\n");
for (j=1; j<=max; j++)
{
    printf("\n%s , %s", member[j].memnam.sur,
    member[j].memnam.frn);
    printf("\nBorn - ");
    printf("%d-%d", member[j].birth.day, memb
    er[j].birth.month);
    printf("-%d", member[j].birth.year);
    printf("\n%- %s\n", "Phone No.", member[j
    ].phone);
}
}
atoi(s)
char *s;
{
    static int c,value,sign;
    while(isspace(*s))++s;
    value=0;
    sign=1;
    if (*s=='-'){++s;sign=-1;}
    else if (*s=='+') ++s;
    while(isdigit(c=*s++))value=10*value+c-'
    0';
    return sign*value;
}

```

Using nested structures. This will be illustrated further in Chapter 9.

FIGURE 6.7

The main structure now is of type **person**, but it now contains the sub-structures **name** and **dob**. The structure variable **memnam** is of type **name**, and **birth** is of type **dob**, both of which must be defined as structures **before** the main structure can be defined. Structure **name** is defined as consisting of **sur** and **frn**, both arrays of **char**. Remember that you can't use **for** for forename, because this is a reserved word. The structure **dob** consists of **day**, **month** and **year**, all integers. These ranges would, in a working program, be checked each time an item was entered.

Though the structures are nested, we don't necessarily need nested loops to read items into the structures, or out of them. For the sake of simplicity, all stages except entry and printing have been omitted from this program. The entry function starts early in the main program, and the important feature here is that the reading line uses the full title for each field and subfield. For the first **member**, 1 in this case, we need the surname. This has to be specified as:

```
member[j].memnam.sur
```

using the main structure title, the substructure title (**memnam**) and the field title of **sur**. Each name and number is entered in this way, using a string entry function which is modelled after **fgets()** so as to restrict the number of characters entered. Remember that the number in this function must be one more than the number of characters, and that a string will always contain a zero character. This is why the day of the month, for example, which has two digits at most, needs a count number of 4 in **getstr**. Following these entries, the phone number is obtained. Note that this is put into string form. A telephone number is most unlikely to be expressible as an integer, and because it may contain dashes (as in 0999-1123-212), it cannot be expressed as a real (float) number either when this becomes possible. The way in which nested structures are printed out is illustrated in the lines following the printing of the title LIST. The loop construction is just as it was before, and the items are specified in the same way. Note how the date of birth items have been fielded so as to make the date look better on the screen. This example shows the most awkward type of sub-structure identification, in which there are several subfields of one main field.

Chapter 7

More about pointers.

We have made some use of pointers in programs and program functions, but the subject so far has really only been introduced. If you look through the routines in the library, you will find that practically all of the standard routines make considerable use of pointers, rather more than we have done so far. The intelligent use of pointers can make a lot of apparently difficult program actions become relatively simple to achieve. A good reason for leaving detailed discussion of pointers until later in this book, however, is that the careless use of pointers can make a program unworkable. In many respects, the use of pointers in 'C' is rather like the use of GOTO in BASIC – it can make a lot possible, but that can include a lot that you don't want.

Before we start on extended pointer exploration, then, recall for a few moments what a pointer is. A pointer is a number which locates a piece of data. A pointer can be defined as a pointer to an **int**, **char** or any other data type, simple or complex. If the data type is a simple one, the pointer is the number which gives the location of the first byte of that data. If the data type is complex, like an array or a structure, then the pointer gives the address of the start of the array or structure. If we want to make use of a pointer, we must declare its name and also assign it. We can carry out actions on pointers that include incrementing and decrementing, addition and subtraction of integers, comparison of pointers, and the subtraction of one pointer from another (only for pointers of the same type). The valuable feature of pointer arithmetic is that 'C' makes automatic allowance for the size of data. If you have an array of characters, for example, then you can define and assign **pc** as a pointer to the first character. Incrementing this pointer, either by using **++** or by adding 1, will get a pointer to the next character. Since each character takes up just one byte of memory, this isn't exactly surprising. If you have an array of integers, however, in which each integer uses two bytes of memory, then changing the pointer by using **++** or by adding 1 will still get the next integer, *even though the pointer has to change by two bytes this time*. This is extremely valuable, because it means that you don't continually have to be worrying about the numbers that you add to pointers. You can, of course, add numbers greater than 1 if you want to get hold of other parts of an array.

```

main()
{
static char nam[]="Sinclair";
static int data[]={1956,1966,1983};
char *pc;
int *pn;
pc=nam;
pn=data;
write(pc,8);
dates(pn,3);
}
write(p,n)
char *p;
int n;
{
int j;
rawout(12);
rawout(10);
for (j=1;j<=n;j++)
putchar(*p++);
}
dates(p,n)
int *p,n;
{
int j;
for (j=1;j<=n;j++)
printf("\n%d",*p++);
}

```

Passing pointers to functions to be used in the functions.

FIGURE 7.1

The use of pointers in this way, along with passing pointer to functions, is illustrated in Figure 7.1. In this example, two arrays have been declared and initialised. One is an array of characters, the other is an array of integers. The assignments **pc=nam** and **pn=data** will make the pointers point to the start of each array. Note that this type of assignment is legal because the name of an array is also the value of its pointer. The important difference between the pointer that we assign and the name is that the name is a **fixed pointer**. In other words we can assign **pc=nam**, because **pc** is a pointer variable, but we *cannot* assign **nam=pc** because **nam** is a fixed amount, the pointer for the start of an array, which cannot be altered except by assigning another array. This important difference is not always well emphasised in books. We could also, incidentally, assign directly to the pointers, not using the array names at all, and this is something that we'll do later. Once the pointers have been assigned, we can use them in function calls. Two function calls are shown, one to **write(p,n)** which will print a string of **n** characters pointed to by **p**, and **dates(p,n)** which will print **n** dates, one on each line, pointed to by **p**.

The real meat of pointer use is now contained in the function definitions, and we'll look at **write** first. The header contains the parameters **p,n** and these will have to be declared before the first curly bracket of the function, because these are the variables that are passed to the function. Since **p** points to a character, it is declared as such, and **n** is an integer. Remember that these names are completely local to the function, we can change them without altering the quantities that are stored as **pc, pn, n** in the main program. Following the first curly bracket, the integer **j** is declared for the loop, and the loop uses **putchar()** to print a character on the screen. Function **putchar** is built-in to HiSoft C, and so it doesn't need to be typed in. The character that **putchar** uses is ***p**, the character that pointer **p** points to. At the start of the loop, **p** takes the same value as **pc**, because this was the value passed to it. In the **putchar()** statement, however, we use ***p++** so that the value of **p** is incremented by one character position *after* the character has been printed. This will ensure that the next character is fetched when the loop goes around again. The **dates** function behaves in an almost identical way, but **printf** has been used to ensure that the date is printed in the form of an integer number. Once again, using ***p++** ensures the correct next number, though this time the memory is being incremented by two units instead of just one. We could make the action of the **write** routine more elegant and more useful by making it write all characters to the '**\0**' character.

```
write(p)
char *p;
{
  rawout(12);
  rawout(10);
  while (*p!='\0')
    putchar(*p++);
}
```

Simplifying the **write()** functions.

FIGURE 7.2

This modification is illustrated in Figure 7.2. Using this function now requires only the pointer to be passed, not the number of characters.

So far, so good. If you want to pass a pointer to a single integer or character, you must use the pointer-finding symbol **&**, which has been illustrated previously. This is particularly important when functions such as **scanf** are used. Generally, however, the main use of pointers is in connection with arrays because this is one of the ways that arrays can be manipulated as a whole. As an example, take a look at the program in Figure 7.3.

```

main()
{
static char nam[]="Sinclair";
static char town[]="Watford";
char *pc,*pd;
pc=nam;
pd=town;
printf("\n%d,%d",pc,pd);
printf("\n%s %s","Name is",pc);
printf("\n%s %s","Town is",pd);
exchange(&pc,&pd);
printf("\n%d,%d",pc,pd);
printf("\n%s %s","Name is",pc);
printf("\n%s %s","Town is",pd);
}

exchange(x,y)
int *x,*y;
{
int tmp;
tmp = *x;
*x=*y;
*y=tmp;
}

```

Swapping string pointers so as to interchange the strings.

FIGURE 7.3

This contains a function **exchange** which will swap two strings of different lengths, by swapping their pointers. Now it's important to realise that only the pointers are swapped, and the strings remain assigned to their original names. If we print out the string names before and after swapping, there will be *no change*. If we print out using the pointers, however, the swap will be obvious. The moral, then, is to work with pointers at all times if you are going to make such changes. The routine of Figure 7.3 declares and assigns two names, and the pointers are declared and then assigned. The **printf** lines then show what the pointer addresses are. If it still unnerves you to see these as negative numbers, use **"%u"** in place of **"%d"** in the **printf** lines for printing the pointers. The first printing shows the pointer numbers and the names in the correct order. Following the **exchange** function, however, the pointer numbers are swapped, and what they point to is also exchanged. This appears only if we **printf** the pointers **pc**, **pd**, not the names **nam** and **town** which do not change. Note that the **printf** line uses **pc,pd** and not ***pc,*pd**. Once again, this is because the name of an array is the pointer to its starting address. In this context, quantities such as ***pc,*pd** are meaningless. The pointer exchange is carried out using **pointers to the pointers**. The pointers to the strings are simply two numbers which are stored in the memory. To exchange them in a function we need to find where these pointer numbers are, and we can do this by finding their own pointers. This is done by calling function **exchange** with parameters **&pc**, **&pd**, which are the pointers to **pc**, **pd** respectively.

These pointers are passed to the function as numbers **x** and **y**, and defined as pointers to integers, which they are. The integers that they are pointing to are the pointers **pc**, **pd**. We then use these pointers to exchange the values of **pc** and **pd**. We cannot simply exchange **pc** and **pd** in a function, because the function works with local values only. At the end of the function, any quantities that are passed to the function are restored to normal. If we work with pointers, however, we can make permanent alterations in anything that these pointers point to. In this case, the quantities that we are working with are temporary values **x** and **y**. These are manipulated so as to exchange **pc** and **pd**, pointers which have not been directly passed to the function. Using pointers in this indirect way is the only method by which a function can make changes in a number of parameters. The routine carries out the swap, and when the main program takes over again you can see that the pointers have been swapped. The important feature here is that a pointer is a two-byte number. You can swap pointers like this around as much as you like, and the action is quick and easy. It's certainly not so easy, and definitely not so fast to try to swop the actual contents of strings around. You wouldn't be advised to try it on strings of unequal defined lengths, either, but when you work with pointers all things are possible. If, incidentally, you want to find where the pointers to the pointers are stored, add a line:

```
printf("\ n%u,%u",x,y);
```

just following the **int tmp** declaration in the function.

Arrays of pointers.

An array of pointers is a method of locating data which is often more useful than other types of arrays. It would be rather pointless (sorry!) to use an array of pointers to integers, because it's simpler to use an array of integers, and it would take less space. Arrays of pointers come into their own when they are used to refer to arrays of arrays. An array of strings, for example, consists of an array of arrays of characters. A useful alternative to a string array formed in the way that we have used previously, then, is an array of pointers to strings. As we have seen, this allows for actions such as exchanging to be carried out. To form and make use of such an array of pointers we have to know what syntax has to be used to refer to pointer arrays. Figure 7.4 shows a simple start to the idea.

```
main()
{
static char s1[]="Xerography";
static char s2[]="Alteration";
static char s3[]="Middle";
char *ptr[3];
int j;
ptr[0]=s1;
ptr[1]=s2;
ptr[2]=s3;
for (j=0; j<=2; j++)
printf("\n%s",ptr[j]);
}
```

Using a pointer array as a string array.

FIGURE 7.4

Three strings have been declared and initialised. They have been declared as **static** because otherwise initialisation cannot be carried out on the same step. An array of pointers is then declared, using **ptr[3]**, and the strings are assigned to the pointers. In a 'real-life' program, it's likely that you would use a **scanf** function to get the strings from the keyboard, so that the assignment would be direct to the pointer array elements. The program then prints the strings in order of assignment.

Now this is all very tame stuff, but you can see that the use of arrays of pointers can lead to a lot of interesting actions. It's as easy to select one string in this way as in a conventional string array, and it's much easier to swap pointers round so that the string arrangement is different.

```
main()
{
    static char s1[]="Xerography";
    static char s2[]="Alteration";
    static char s3[]="Middle";
    char *ptr[3];
    int p;
    int j;
    ptr[0]=s1;
    ptr[1]=s2;
    ptr[2]=s3;
    for (j=0; j<=2; j++)
        printf("\n%s",ptr[j]);
    alter(&ptr[0],&ptr[1],&ptr[2]);
    for (j=0; j<=2; j++)
        printf("\n%s",ptr[j]);
}

alter(x,y,z)
int *x,*y,*z;
{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;
    tmp=*y;
    *y=*z;
    *z=tmp;
}
```

Using a pointer array to swap string positions.

FIGURE 7.5

Figure 7.5 shows this done, using a simple swap routine rather than the complications of a full-scale sort for so few items. Note that, contrary to what you might expect, you have to use the address sign, `&`, in front of the pointer array values in order to pass the values correctly to the function. If you omit the `&` signs, you will find that the first two letters are exchanged, but no others. This is not a standard action of 'C', and it's not easy to see why *two* characters are swapped. One, or all, I could understand, but two? Another point to note is that the use of a function will successfully result in obtaining pointers to the pointers, but this is not so simple if you want to carry it out in the main part of the program. The reason is that quantities such as `ptr[0]` are defined as string pointers, and you can't obtain pointers to them in a straightforward way. The only simple assignment you are allowed to make is of another pointer to a string. You can get around this restriction by using a statement `cast` which is very poorly illustrated in most books on 'C'. The use of `cast` is to make a quantity become of a specified type, and the syntax is `cast(type)quantity`. This doesn't illustrate how `cast` is used, however, quite so dramatically as Figure 7.6.

```
main()
{
static char s1[]="Xerography";
static char s2[]="Alteration";
static char s3[]="Middle";
static char *ptr[3];
static int p,*s;
int j;
ptr[0]=s1;
ptr[1]=s2;
ptr[2]=s3;
s=&(cast (int) ptr[0]);
for (j=0; j<=2; j++)
{
printf ("\n%u,%u",s,*s);
printf ("\n%s",*s++);
}
}
```

Using `cast` to convert a pointer into unsigned form.

FIGURE 7.6

In this example, the strings have been printed in quite a different way that illustrates the usefulness of pointers to pointers. The line:

```
s=&(cast (int) ptr[0]);
```

will have the effect of making the quantity `ptr[0]` temporarily into an integer, and then taking the address of this integer and assigning it to integer pointer `s`. It is possible to do this in two steps, such as:

```
p=cast(int)ptr[0];s=&p;
```

but when you do this the addresses will be displaced, because of the extra integer, **p**. If you use this method, you will find that the output is decidedly peculiar, with the first string being printed twice! Using the correct **cast** expression allows **s** to carry the address of **ptr[0]**, and the printing loop can now make use of ***s++** both to print the value and to increment to the next string. This is simple because the string pointers are held at consecutive addresses.

Another aspect of the use of pointers with string lists is how easily an item can be located. This makes for very efficient and short routines for such actions as finding the day of the week from a number.

```

char *getname(c)
int c;
{
    static char *day[]={
        "no such day", "Monday", "Tuesday", "Wedne
sday", "Thursday", "Friday", "Saturday",
        "Sunday"};
    return((c<1 || c>7)?day[0]:day[c]);
}
main()
{
    char c;
    rawout(12);
    printf("\nDay number, please\n");
    c=(getchar()-48);
    printf("\n%s",getname(c));
}

```

Using a string pointer array, initialised at the same time as being declared.

FIGURE 7.7

An illustration of this use is shown in Figure 7.7, which also shows the initialisation of a set of strings in a function. The function is of type **char** and it will return a pointer because this is what the name of one item will be, one of the array of pointers. The header will put in an integer which will consist of a number in the range 0 to 9. A number such as 10 will count as 1, because only the first character will be accepted by the main program call to **getchar**. This number is declared and treated as an integer in the function, and the pointer array ***day** is declared and (since it is static) assigned. The assignment of the zero position is made to a message. After this, the last line of the function returns the selected string. The variable **c** is used as a selector in the line:

```
return((c<1 || c>7)?day[0]:day[c]);
```

so that if **c** is less than 1 or more than 7, the string **day[0]** will be returned, giving the message 'no such day'. For numbers between 1 and 7 inclusive, the correct day of the week is returned counting Monday as day 1.

Pointers to functions.

One of the restrictions about the use of functions is that you can't use a function name as a parameter to a function. It would be useful, for example, if a function could specify in its header which of a number of alternative functions was to be used inside it. This action is, however, possible provided that the name which is passed is treated as a pointer to the function that is wanted. Once again, the use of a pointer makes a very useful and curious action possible.

```
main()
{
static char test[]="TeSt";
int down(),up();
int j;
printf("\n%s",test);
for (j=0;j<=3;j++)
{
if (test[j]<91) test[j]=redo (test[j],u
p);
else if (test[j]>96)test[j]= redo(test[
j],down);
}
printf("\n%s",test);
}
redo(c,change)
char c;
int (*change)();
{
c=(*change)(c);
return(c);
}
up(s)
char s;
{
s+=32;
return(s);
}
down(s)
char s;
{
s-=32;
return(s);
}
```

Passing a pointer to a **function** to allow a choice of functions.

FIGURE 7.8

Figure 7.8 illustrates this way of allowing functions to pass other functions as parameters. The example is a very simple one, which means that what it does could very easily be done by simpler methods. The point, however, is that its only by looking at fairly simple examples that you can disentangle the important features from their surroundings. In this example, the program is provided with a word which is typed partly in upper-case and partly in lower-case, and the aim is to reverse the case of each letter.

The main program is simple enough, but the declarations have to be watched. The functions that are going to be passed as parameters have to be declared at the start of this main program, and they are called **down** and **up**. Each of them will return an integer, so that they can be placed at the end of the listing without any need for references forward. Following the declarations, the program starts a loop in which each character of the word is selected and passed as a parameter to a function called **redo**. This function does *not* need to be declared in the main program, and it will be designed to return the character rather than altering a pointer. Because it returns a character, its form is:

```
character=function(parameters)
```

and in the example, the two different calls to **redo** are placed in two test lines. If the character that is being dealt with has an ASCII code of less than 91, then it is an upper-case character, and function **up** must be called within the **redo** function. This is done by using the call:

```
test[j]=redo(test[j],up);
```

in which we want to alter **test[j]** by equating it to the character returned by **redo**. We also want **redo** to make use of function **up**, and this function name is put into the parameter list for **redo**. Note that we use **up**, and not **&up**. This is because 'C' takes the name of a function, like the name of an array, as a pointer to where the function starts. The alternative call is made if **test[j]** is greater than 91, when the call to **redo** makes use of the function **down**.

The next step is to look at function **redo** to see how the alternative functions **up** and **down** are used. The header for **redo** uses parameters (**c,change**) in which **c** is a character code and **change** represents the pointer to a function passed to **redo**. This function represented by **change** has to be declared before opening the curly bracket on **redo**, and it's declared as:

```
int (*change)();
```

– a function which returns an integer and whose (temporary) name is pointed to by **change**. You *must* use the name as a pointer here, with the asterisk, and the brackets surrounding the function name, and a separate set of brackets as you have with any function. The **redo** action then consists only of calling the function that has been passed to **redo**, and returning the correct character. Once again, however, the syntax is important. The temporary name must be used as a pointer and enclosed in brackets, with its parameter **c** in separate brackets following (***change**). Since this function needs to return a character, we use:

```
c=(*change)(c);  
return(c);
```


to ensure the return of a suitably changed character. The effect of all this will be to pass the pointer to one of the functions **up** or **down**, and execute the action of that function from within **redo**. It's not exactly the kind of thing that you can do in BASIC, and it's one of the many features that makes 'C' such a very powerful language for programming. The functions **up** and **down** are written conventionally, but the lines **return(s)** are, in fact, redundant. They have been put in as a reminder that each function returns something, but they can be deleted because the **return(c)** inside **redo** carries out the returning action.

Passing function names in this way is particularly useful when you want to use a function on different types of data. One very common type of action is sorting lists of different items. This can call for the use of different comparison functions (one for numbers, one for strings) and different exchange functions (once again, differing for numbers and strings) but with the same basic action, such as the Shell sort.

```
/* Sorting function - a Shell sort */
void qsort(list, num_items, size, cmp_func)
char *list;
int num_items, size;
int (*cmp_func)();
{
    static unsigned gap, byte_gap, i;
    static char *p;

    for (gap = num_items >> 1; gap > 0;
        gap >>= 1)
    {
        byte_gap = gap * size;
        for (i = gap; i < num_items; ++i)
        {
            for (p = list + i * size - byte_gap; p >= list; p -= byte_gap)
            {
                if ((*cmp_func)(p, p + byte_gap) <= 0) break;
                swap(p, p + byte_gap, size);
            }
        }
    }
}
```

The Shell sort from the HiSoft library, courtesy of HiSoft.

FIGURE 7.9

You can obtain this by using `g,,stdio.lib`, and you can then separate out routines and record them for use as and when you want. In this particular example, four variables are passed, one of which is a function `cmp_func`. This will be the function that compares items in an array. The start of the array is pointed to by `list`, and the size of the items must be constant. The number of items and size is passed as integers, and the comparison method is a function.

The action of calling the sort is better illustrated by an example, in Figure 7.10.

```

main()
{
    int numcmp(), strcmp();
    static int x[]={3,17,4,21,16};
    static char nam[5][10]={"zebra","delta",
        "whisky","juliet","echo"};
    int j;
    for (j=0; j<=4; j++)
        printf("\n%d----%s",x[j],nam[j]);
    printf("\n");
    qsort(x,5,2,numcmp);
    qsort(nam,5,10,strcmp);
    for (j=0; j<=4; j++)
        printf("\n%d----%s",x[j],nam[j]);
}

int qsort(list, num_items, size, cmp_func)
char *list;
int num_items, size;
int (*cmp_func)();
{
    static unsigned gap, byte_gap, i;
    static char *p;
    for (gap = num_items >> 1; gap > 0;
gap >>= 1)
    {
        byte_gap = gap * size;
        for (i = gap; i < num_items; ++i)
            for (p = list + i * size - byte_gap; p >= list; p -= byte_gap)
                if ((*cmp_func)(p, p + byte_gap) <= 0) break;
                swap(p, p + byte_gap, size);
    }
}

```

```

int strcmp(s, t)
  char *s, *t;
{
  while (*s == *t)
    {
      if (! *s) return 0;
      ++s; ++t;
    }
  return *s - *t;
}
int numcmp(x,y)
int *x,*y;
{
  if (*x>*y)return(1);
  else if (*x<*y) return (-1);
  else return(0);
}

```

Making use of the library function `qsort`.

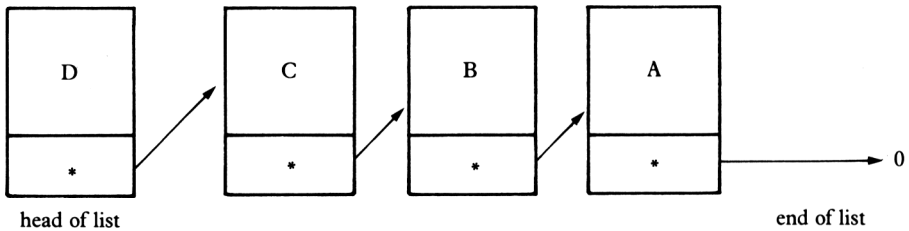
FIGURE 7.10

In this example, two arrays are declared and initialised. One is an array of integers, the other an array of strings which is declared as a two-dimensional array and filled with words. Note the order of the subscript numbers, which is **[number of words][maximum size]**. These arrays are printed side by side to show the order, and then two calls are made to the function `qsort`. This has been loaded in from the library with only one change. The change is to make the type of function `int` rather than `void`. The library declares early on that `void` is identical to `int`, and the word is used when whatever a function returns is unimportant. The `strcmp` function is also taken from the library, but the `numcmp` function has been added by me. When a `qsort` call is made, then, its parameters are the start of the array, the number of items in the array, the size of each item, and the comparison function. The pointer for the start of any array is always the array name, and the number of items is always five in this example. The sizes of items are 2 for integers, which store in two bytes, and 10 for strings, since we have defined strings of ten characters. Remember that this figure of ten characters includes the ending zero of a string. If when you print your strings you see two or more joined, this indicates that the zero has been wiped out. The comparison functions will be `strcmp` for the strings, and `numcmp` for the numbers.

All the actions in this program are by now reasonably familiar, but the function `numcmp` is new. The `qsort` routine makes use of pointers, and what is passed to `numcmp` is a pair of pointers to integers. These are declared, and then the main action consists of comparing the integers that `x` and `y` point to. These comparisons return numbers 1, -1, or 0 according to whether the order is incorrect, correct or correct. The order will be correct if the two numbers are in order, or if they are equal, which is why the three possibilities exist. When the routines run, then, you will see the lists in their original unsorted form, and then in their sorted form.

Linked lists.

Suppose that you defined a structure which consisted of an integer number and a pointer. Now the most important feature of a pointer is that it can be made to point to something that may be anywhere in the memory of the computer. Because this is possible, we can make the pointer in the structure point to the next structure, even if this means a structure which is not the next one that you enter, or even the previous one. If you make up a set of structures like this, you don't need an array. Each structure contains a pointer to the next structure so that if you can locate the first structure, you can get to any other, swinging like the legendary Tarzan on ropes of pointers from structure to structure.

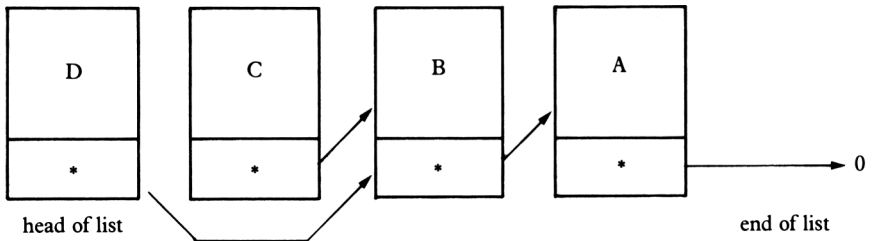


A linked list in diagram form. This is the form of BASIC program lines.

FIGURE 7.11

Figure 7.11 shows in diagram form what this is all about. A sequence like this is called a 'linked list'.

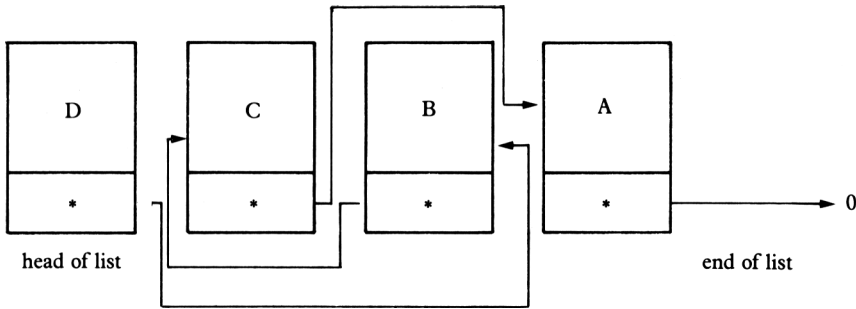
What advantages would such a set of structures have? Well for one thing, it becomes very easy to 'delete' a structure. All you need to do is to alter the pointer that points to the item and make it point to the next one instead (Figure 7.12).



Deleting a structure from a linked list.

FIGURE 7.12

Having coped with that idea, how would you reverse the order of two structures?



Reversing the order of two members of a linked list.

FIGURE 7.13

Figure 7.13 shows the principle in diagram form, requiring three pointers to be changed. What we have to do now is to see how some of this paper talk can be transferred into a working program.

```

struct record{
    int daily;
    struct record *next;
}cash,*first,*p;
typedef struct record *rec_p;
main()
{
    int j,x;
    p=cast(rec_p)calloc(100,sizeof(struct
record));
    first=NULL;
    do{
        printf("\nToday's number - ");
        scanf(" %d",&j);
        p->daily=j;
        p->next=first;
        first=p;
        p++;
    } while (j!=0);
    rawout(12);
    x=rawin();
    j=1;
    p=first;
    while(p!=NULL)
    {
        printf("\n%d---%d",j,p->daily);
        p=p->next;
        j++;
    }
}

```

A simple linked list example. This does not show the supporting functions, and will not run unless these are added or put in with `#include ?stdio.lib?`.

FIGURE 7.14

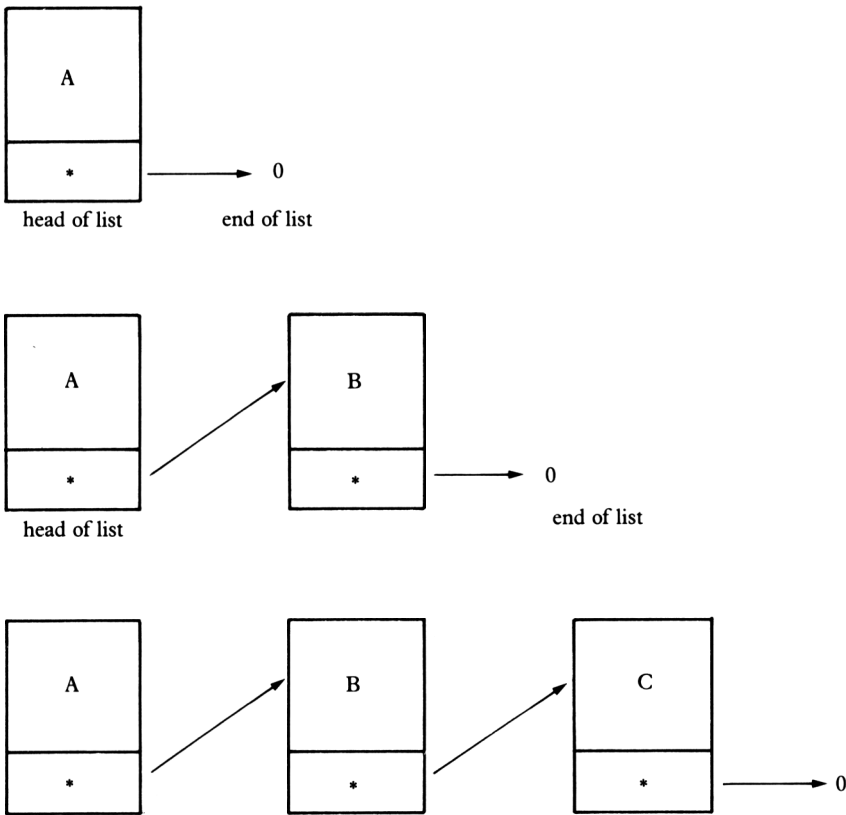
Figure 7.14 shows a typical example which is deceptively simple. Only the main part of the program has been shown here, because when all the necessary functions are added to make it run, the result is rather massive. In this part, then, we define a structure called **record**. This structure contains an integer **daily**, and a pointer **next**. What makes **next** rather different from any pointer we have looked at so far is that it's a pointer to type **record**. In other words, part of this structure is a pointer to another structure of the same type. Variable **cash** is then defined as a structure of type **record** which contains **daily**, an integer number, and **next**, a pointer. The variables **first** and **p** are also defined as pointers to a **record**, then the program itself starts with **main()**. This starts by declaring **j** and **x** as integers, and then finding a value for the quantity **p**. We need **p** as a pointer to where each structure is going to be stored, and to get this free space we use a function **calloc(number, size)**. The parameters to **calloc** are the number of structures that we would want to use, and the size of each structure. Rather than count up the size of a structure, we can use **sizeof** to get this number. The snag in this action is that **calloc** returns a character pointer, and we want a pointer to type **record**. This is arranged by two important lines. One is the **typedef struct record *rec__p** which follows the declaration of the structure. We haven't used **typedef** before, because it's main use is in situations like this along with **cast**. The statement **typedef** allows us to declare that a word represents a data type. It doesn't allow us to create any new data types, but it *can* define a type that will be accepted by the **cast** statement. In this example, the word **rec__p** is being defined as meaning a pointer to type **record**, the structure. By using **rec__p** in the **cast** statement, therefore, we force the value of the pointer that is returned by **calloc** to be of the same type as a pointer to the structure. You may think that all of this changing of variable type for a set of numbers that are all integers is rather tedious, but programming can be a lot more tedious in a language which doesn't allow conversions!

The next step is to make **first** point to NULL. There are no values to point to yet, so this pointer points nowhere. A loop is then set up. The preset limit to size for this set of structures is set by the value of size that was used in **calloc** and entry can continue until you enter a figure of zero, or until the memory is full. You get no warning if you exceed the limit that has been set by **calloc**, and if you do exceed it you may suffer no ill effects, or you may find that the whole program crashes, and your data with it. The freedom that you have to do as you wish with 'C' is paid for by the need to ensure that your program can't do anything silly.

Now what happens in the loop is vital to the way that the list is constructed, and you need to follow it very carefully. I think it's a lot easier if we can think of numbers for the pointers, and so I'll assume that the pointer NULL is zero (true), and that the other pointers will be 40974, 40978, 40982... and so on (possibly true, depending on machine, and it helps you to understand it). Let's take a walk through the first loop round. The value of pointer **p** has already been allocated by **calloc**, and its type is pointer to **record**, so that we can use quantities such as **p->daily** in the same sense as **cash.daily**. We can imagine that **p** carries the number 40974. This will now be used to store the cash amount, **p->daily**, obtained as integer **j** from the **scanf** line. Notice, incidentally, the blank space preceding the **%d** in **scanf**. This is put in to ensure that the (RETURN)/(ENTER) character does not cause an endless loop. The pointer quantity **p->next** for this structure is now made equal to **first**, which is NULL. This is the way of signalling that there is no following structure.

Pointer **first** is then made equal to **p**, assumed to be 40974. The value of **p** is then incremented using **p++**. This makes **p** change from 40974 to 40978, because the structure contains one integer (two bytes) and one pointer (also two bytes), a total of four bytes.

So much for the first loop. What happens in the second? We pick a new **p** allocation, assumed 40978 this time. Once again, we store a cash quantity, and the **p->next** pointer is this time made equal to 40974, the pointer to the **previous** entry. Pointer **first** is now made equal to the current value of **p**, which is 40978. If you look at these steps in diagrammatic form, Figure 7.15, you can see that the list is not growing in the way that you might expect.



The steps in the formation of the list.

FIGURE 7.15

The 'top' of the list is zero, followed by the item that we entered last of all. Its pointer always points to the next one down, the previous item. The list ends with the one that points to **the first entry**. If any more proof is needed, take a look at what the last part of the program,

following the zero entry, prints out. Enter items like 11, 22, 33 and so on that you can recognise. When you see the listing it will show 1-0, which is the zero entry that you used to close the list. After that your other values follow in **reverse** order of entry, with the most recently entered item at the top of the list and the first item that you entered at the bottom. The word **next** can be rather confusing in program examples like this. It certainly means the next item in the list, but it's the next one **back** simply because there isn't a next one forward until you create one!

```

#define TRUE 1
#define FALSE 0
#define NULL 0
#define ERROR -1
typedef char* __char_ptr;
struct _header
{
    struct _header * _ptr;
    unsigned _size;
};
typedef struct _header  HEADER, * HEADER_PTR;
HEADER _base, *_allocp;
#define HEAPSIZ 1000
char sbrk(n)
unsigned n;
{
    static char *p, heap[HEAPSIZ], *heap_ptr=
    heap;
    if (heap_ptr+n>heap+HEAPSIZ)return ERROR
    ;
    p=heap_ptr;
    heap_ptr+=n;
    return p;
}
char *calloc(n, size)
    unsigned n, size;
{
    static HEADER *p, *q;
    static unsigned nbytes;
    nbytes = (n * size + (sizeof(HEADER) -
    1)) / sizeof(HEADER) + 1;
    if ((q = _allocp) == NULL) /* no free
    list */
        {
            _base._ptr = _allocp = q = &_amp;_base
        ;
            _base._size = 0;
        }
}

```



```

    p = q->_ptr;
    while (TRUE)
    {
        if (p->_size >= nbytes) /* big enough */
        {
            if (p->_size == nbytes) q->_ptr = p->_ptr; /* just right size */
            else
            {
                /* split block and allocate tail */
                p->_size -= nbytes;
                p          += p->_size;
                p->_size  = nbytes;
            }
            _allocp = q;
            return cast(__char_ptr) (p+1);
        }
        if (p == _allocp) /* wrapped around free list */
        {
            if ((p = cast(HEADER_PTR) sbrk(nbytes * sizeof(HEADER))) == ERROR)
                return NULL;
            p->_size = nbytes;
            free(p+1);
            p = _allocp;
        }
        q = p;
        p = p->_ptr;
    } /* end while TRUE */
}

struct record{
    int daily;
    struct record *next;
}cash,*first,*p;
typedef struct record *rec_p;
main()
{
    int j,x;
    p=cast(rec_p)calloc(100,sizeof(struct record));
    first=NULL;

```

```

do{
    printf("\nToday's number - ");
    scanf(" %d",&j);
    p->daily=j;
    p->next=first;
    first=p;
    p++;
} while (j!=0);
rawout(12);
x=rawin();
j=1;
p=first;
while(p!=NULL)
{
    printf("\n%d---%d",j,p->daily);
    p=p->next;
    j++;
}
free(p);
}
int free(block)
    char *block;
{
    static HEADER *p, *q;
    p = cast(HEADER_PTR) block - 1;
    for (q = _allocp; !(p > q && p < q->
_ptr); q = q->_ptr)
        if (q >= q->_ptr && (p > q || p
< q->_ptr)) break;
    if (p + p->_size == q->_ptr)
    {
        p->_size += q->_ptr->_size;
        p->_ptr = q->_ptr->_ptr;
    }
    else p->_ptr = q->_ptr;

    if (q + q->_size == p)
    {
        q->_size += p->_size;
        q->_ptr = p->_ptr;
    }
    else q->_ptr = p;
    _allocp = q;
}

```

The complete listing for the linked list program, with all the library functions, courtesy of HiSoft.

FIGURE 7.16

Figure 7.16 shows the complete listing for the linked-list, using all the library listings that are included. In these, the word **void** has been replaced by **int**. Don't be downhearted by the look of a listing like this, because as you can see, the main program, which is the bit that *you* write, is quite small. The routines from the library can be taken from the disc and added in, with no typing required. You can do this by making a disc in which each library routine is separately filed. You will find this very useful because it allows you to make up programs without the need to use **#include ?stdio.lib?**, which slows down compiling considerably. You can load in your library routines when you compile by using **#include.filename** at the start of the program.

Could we, as a matter of interest, make a list in a different way? When you think about it, there's no reason why you shouldn't. At any point in a list you can direct a pointer to the next item or to the previous item simply by incrementing or decrementing the **p** number. This makes it possible to construct what are called 'double-linked' lists with a pointer in each direction, but programming of that sort is beyond the scope of this book, and if you need to use it you probably need a lot more memory for your program than you'll find in the smaller Amstrad machines. What is more important at this point, then, is learning how to operate on the lists so that you can search through a list.

```

struct record{
    int daily;
    struct record *next;
}cash,*first,*p;
typedef struct record *rec_p;
main()
{
    int j,x,y;
    p=cast(rec_p)calloc(5,sizeof(struct re
cord));
    first=NULL;
    x=0;
    do{
        x++;
        printf("\n%s%d%s","Item ",x," is - ")
;
        scanf(" %d",&j);
        p->daily=j;
        p->next=first;
        first=p;
        p++;
    } while (j!=0);
    rawout(12);
    --x;
    do{
        printf("\nplease select item number.");
        printf("\n%s%d%s","(range-1 to ",x,")");
        p=first;

```

```

scanf(" %d",&j);
if (j>x||j<1) continue;
else{
    printf("\n%s%d%s","Item ",j," is ");
    j=x+1-j;
    for (y=1;y<=j;y++)
        p=p->next;
    }
    printf("%d",p->daily);
}while (TRUE);

```

Searching through a linked list for an item.

FIGURE 7.17

Figure 7.17 shows how you can search through a list for an item. This is the main part of the program only, and you will need the same additional functions as the previous listing. You need to know, of course, what you are looking for, whether it's the item with a value of 64 or the first one whose value is less than 9 or whatever criterion you adopt. You need not work, of course, with integer number values in the structures, as long as each structure contains some data that you want to use along with a pointer to the next structure. In this example, the same structures have been used to save extra typing. The aim is to find the entry for a given identification number, but you could, of course, easily adapt the program to find whatever feature you wanted of the structure you had defined. The item-finding part has been put into a loop so that you can try the options of a number that you know is genuine and one that you know is not in the list. You can escape in the usual way by pressing the (ESC) key when you see the 'please select item number' message.

There are a few alterations to both parts of the main program this time. An extra integer, *y*, is declared, and integer *x* is used to count the items as they are entered. When entry is completed, this number is corrected to remove the zero entry which terminated the entry routine. This also makes it unnecessary to test for the pointer becoming equal to NULL, though it is better practice to do this also. Following the clearing of the screen and the decrementing of counter *x*, the endless loop starts. In this loop you are asked to provide an item number, which is tested for range. If the item number is in the correct range, the corresponding item is found. This is not entirely straightforward because of the 'upside-down' nature of the list. The **scanf** function returns the item number as integer *j*, and this is tested for range using **continue** to ensure that the loop will return to its starting point if the number is not in range. For a number that is in range, the next step is to adjust the variable value, using the expression:

$$j=x+1-j$$

to 'invert' the number. If you have 9 items, for example, and you want the third one, then this expression gives $9+1-3=7$, which is the correct position on the list counting from the end at which the last item was entered. The corrected number *j* is then used in a loop to run through the pointers until the correct item has been obtained. The item value is then printed in the usual way.

From this point on, the topic of linked lists starts to get complicated, and you will need to consult specialised texts if you want to see how to insert or delete items. It's advisable to make sure that you have a really firm grip on 'C' programming before you attempt such work, and that's why I shall not pursue the topic in this book.

Chapter 8

String functions.

Compared with BASIC, standard 'C' seems at first sight rather weak on string handling routines. This is particularly true when 'C' is compared to Locomotive BASIC and to some of the later versions of Microsoft BASIC, as used, for example, in the MSX micros and also in the IBM PC. HiSoft C sticks close to the standard in most respects, so that you must rely on library routines to provide the string handling that you have become used to in BASIC. When you have moved from BASIC to 'C', however, this can leave you wondering how to implement actions, such as TAB, which you always took for granted with BASIC. You also have to learn how to use library functions for things like LEFT\$, MID\$ and RIGHT\$, and the HiSoft manual is particularly helpful in this respect. You will find that a few actions, notably the VAL action of BASIC, are performed almost automatically and without effort in 'C'. The problem, in fact, is very often to make sure that any variable is in the form that you want! This chapter will be devoted to the use of the string-handling library functions which help you to bridge the gap. A named 'C' function has the great advantage over a BASIC subroutine in that it is called into action by using its name rather than by the use of a line number. Because of this 'C' functions can be used as if they were new words in the language, which, from your point of view, they are. All of this is greatly helped by the appending action of **g,filename**, which will tack a file on to the end of whatever is in place. You can also use **#include filename** to read in library functions from a disc which has the functions stored separately. The one action that is missed most by ex-BASIC users, however, is assigning a string variable to some text, as **A\$="THIS IS A STRING"**. This is not **very** difficult to implement in 'C', but it requires some planning, unlike its BASIC equivalent. You should try to make use of string **constants**, prepared with **#define** rather than string variables for messages and other items of this sort. Other string assignments can often be tackled by the type of combined declaration and initialisation that we have looked at already.

```

#define TAB "          "
/* eight spaces*/
#define CLS rawout(12)
#define EOF -1
#define NULL 0
char *gets(s)
    char *s;
{
    static int    c;
    static char  *cs;
    cs = s;
    while ((c = getchar())
        !=EOF  &&  c!='\n')
        *cs++ = c;
    *cs = 0;
    return
        ((c== -1 && cs==s) ?
         NULL : s );
}

char *strncpy(s1, s2, n)
    char *s1, *s2;
{
    static char *s, c;
    s = s1;
    c = *s2;
    while (n)
        {
            *s++ = ( c ? ( c = *s2++ ) : 0
        ) ;
            --n;
        }
    return s1;
}

char *spc(s,n)
char *s;
int n;
{
    static char sp[]="          "
    ;
    /* 20 spaces*/
    static char *spc;
    spc=s;
    strncpy(spc,sp,n);
    spc[n]='\0';
    /*needs terminator*/
    return spc;
}

```



```

unsigned strlen(s)
    char *s;
{
    static char *p;
    p = s;
    do ; while (*p++);
    return p-s-1;
}

main()
{
    char s[40];
    CLS;
    printf("\n%s%s%s%s%s%s\n", "A", TAB, "B",
    TAB, "C", TAB, "D");
    printf("\n%s\n", "Your string, please");
    gets(s);
    printf("\n%su%s\n", "consists of ", strl
    en(s), " characters.");
    printf("\n%s%s%s%s%s", "This", spc(s,5),
    "is", spc(s,3), "spc", spc(s,4));
    printf("%s%s%s", "in", spc(s,7), "action");
}

```

A few string actions, including fixed tabbing, **strlen**, and **spc**.

FIGURE 8.1

The set of string actions starts with Figure 8.1. I could have shown all of the string library functions in one long program, but I have split these into sections because it's easier this way to select the actions that you need most frequently. Be warned, though, that these functions are in 'skeleton' form. They are completely lacking in any mugtraps, so that misuse can crash your programs. The aim is to provide simple functions, with no frills, which you can use as starter packs for your own routines. The simplest string action consists of preset tabbing. By defining a constant **TAB** of eight spaces, we can place the word **TAB** into a **printf** statement and automatically space words or numbers. Remember, however, that this does not over-ride any field specifications, and that typing **printf("/n%s%s%6d", "Total", TAB, j);** where **j** is an integer will produce more than eight spaces because of the field number '6'. If you field your numbers correctly, however, this will not be a problem. The use of this preset **TAB**, which can, of course, be of five spaces or any other number you please, gets over a lot of the display problems that ex-BASIC programmers encounter when they start to use 'C'. In this sense, **TAB** is being used more like the **TAB** of a typewriter than the **TAB** of BASIC. The action is closer to the instruction **SPC(8)** which is allowed in some versions of BASIC, or to the way that the comma is used in BASIC. Remember that 'C' can use the field numbers for its tabulation, and these are often much easier to use than the BASIC **TAB** instruction. The **TAB** in this example is produced simply by using **#define**, with **TAB** made equivalent to eight spaces. In addition, **CLS** is defined to be equivalent to **rawout(12)** to clear the screen.

The next action in the list is the **strlen** function. This is a HiSoft library function which produces a value for the number of characters in a string up to the final zero. This is not the same as the total length of the string, because the zero is not counted. The string name is passed as **s**, defined in the header as a pointer. Another pointer, **p**, is then declared and set equal to the value of **s** so that at this instant, both **s** and **p** are pointing to the start of the string. The **do..while** loop then increments **p** until it points to a zero, the end of the string. The length, including the zero, is then **p-s**, and subtracting another 1 gives the length not counting the final zero. A lot of other functions can make use of **strlen**, so that this function is one of our 'bankers', one that you should keep on a disc and not delete from this program unless you are sure that it is not used in later versions.

The function **spc(s,n)** is one that will return a specified number of spaces. This is *not* one of the library functions, though it depends for its effect on a library function which copies one string into another. This function has been written so that it returns a string, and can therefore be included in a **printf** line. It could equally easily have been modified so that it printed the spaces itself, and returned nothing. It's up to you to choose how you want to use a function of this type, but the use of a function that returns something is often more flexible. In this case, the variables **s**, a string to be returned, and integer **n**, the number of spaces, will be passed to the function. These are declared in the header, with **s** as a pointer to the first character of the string. Following the curly brackets, the string **sp** is declared and assigned to twenty spaces. This means that if you try to create more than 20 spaces, you will get very odd results! The pointer **spc** is also declared, and assigned to **s** to pass the string back. The library function **strncpy** is then used to copy the correct number of spaces into the string **spc**. This library function will *not* put in the zero at the end of the string, and this has to be done in a separate line. The string **spc** is then returned.

In the library function ***strncpy(s1,s2,n)**, a number of characters **n** from the string **s2** are copied into a string **s1**. If the string **s2** is short, shorter than 'n' characters, it is copied complete, with its ending zero. If, however, the string **s2** is long and the number **n** selects only part of the string, then the final zero is not copied. You have to be certain before you make any use of this function (and the similar **strcpy**) that the string **s1** has been defined as long enough to take whatever is being copied into it. As usual 'C' lets you do whatever you want to, even if it's likely to crash the whole program! In the **strncpy** routine, the string pointers are declared in the header. The integer **n** is not, because the declaration of an integer in the header is optional in HiSoft C, though it's very desirable to put in the declaration if only as a reminder. You'll find some integers declared, others not declared in the HiSoft C library, depending on whoever wrote the original versions. At the start of the function, pointer **s** to char, and char **c** are declared, then the statement **s=s1** sets the local pointer **s** to the start of the string into which characters are to be copied. Character **c** is then assigned to the first character of the second string, **s2**. A loop then starts which will run as **n** is counted down until the value of **n** reaches zero. In the loop the value assigned to ***s**, the string copy character, will either be the character copied from **s2**, or zero. If the current character **c** is not zero, then the character from **s2** is used, else a zero is put in. Note, however, that the zero is copied only when a zero exists in the second string. The string pointers are incremented, and the number **n** is decremented. If the loop finishes because **n** has been decremented to zero, then no zero is copied. Looking at this loop, incidentally, you might be tempted to rewrite the whole routine in the form that is shown in Figure 8.2.

```

char *strncpy(s1, s2, n)
char *s1, *s2;
{
    static char *s;
    int j;
    s = s1;
    for (j=n; j>0; j--)
        *s++=(j?(*s2++):0);
    return s1;
}

```

A rewrite of **strncpy** which is more suitable for these applications.

FIGURE 8.2

This uses a **for** loop to carry out the action, and has the advantage, for our purposes, of always putting in the terminating zero when the whole string has not been copied. This, of course, is not a replacement for **strncpy** except for this type of use. The more general library routine is the one to use when you want to copy entire strings, and when the end is to be signalled by the zero character normally rather than by the count.

The main program illustrates the use of this 'starter-pack' of string handling functions and functions. The screen is cleared, using the defined CLS, and the letters A, B, C, and D are tabbed across the screen, using the constant TAB. A figure of eight spaces is a particularly convenient one for most purposes, but you might want to use 5 for separating integers particularly if the field size is small. Remember that this TAB, unlike TAB in BASIC, is really a spacing command. In other words, it sets the amount of space between items rather than the absolute position of items on the line. The next three lines illustrate the **strlen** function being used simply to print out the length of a string. This would not normally be of interest but you might want to make use of **strlen** to decide whether a string was too long to use, or in other string handling actions. Finally, the last two lines illustrate the use of **spc**. This, remember, is a function which returns a string so that it can be put into a **printf** statement like any other string.

```

#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *gets(s)
char *s;
{
    static int c;
    static char *cs;

    cs = s;
    while ((c = getchar())
           !=EOF && c!='\n')
        *cs++ = c;
}

```

```

    *cs = 0;
    return
        ((c==-1 && cs==s) ?
         NULL : s );
}
main()
{
    int x,y;
    char s[40];
    CLS;
    printf("\nPlease type your text\n");
    gets(s);
    printf("\nand the x,y, numbers\n");
    scanf(" %d %d",&x,&y);
    /*should test these*/
    printat(x,y,s);
}
printat(x,y,s)
int x,y;
char *s;
{
    rawout(31);
    rawout(x);
    rawout(y);
    printf("%s",s);
}

```

Achieving a PRINTAT action.

FIGURE 8.3

A function which resembles the PRINTAT facility of some computers is illustrated in Figure 8.3. In the BASIC of many computers, PRINTAT is followed by X and Y coordinates and then by a string which is to be printed. This allows a string to be printed **anywhere** on the screen, irrespective of the normal position of the cursor. PRINTAT can be implemented very easily in HiSoft C thanks to the provision of **rawout()** in conjunction with the Amstrad codes, and the PRINTAT action is almost as fast as normal printing. The function makes no use of any other library function, because **rawout** is a built-in function of HiSoft C. The function requires three parameters, the X and Y positions (integers) and the string text. In this example, no attempt has been made to test the values of **x** and **y**, and if you seriously wanted to use this function, you should really include some limits on the numbers. In the main program **gets(s)** is used to get the text, and **scanf** is used to get the numbers. The spaces in the **scanf** specifier portion are deliberate, in case you want to use the program in a loop. You can enter the numbers either by typing one, then (RETURN)/(ENTER), then the other (and (RETURN)/(ENTER)), or by typing the numbers separated by a space and then pressing the (RETURN)/(ENTER) key. This latter method is much quicker, but you have to remember to get out of old BASIC habits – no comma must be used.

The use of **printat()** gives you a lot more freedom over the placing of text on the screen, and saves having to make each positioning of text into a major exercise. Useful additions when you don't necessarily want the complete freedom of **printat** are **vspc** and **cent**, illustrated in Figure 8.4.

```

#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *gets(s)
    char *s;
{
    static int    c;
    static char  *cs;

    cs = s;
    while ((c = getchar())
           !=EOF  &&  c!='\n')
        *cs++ = c;
    *cs = 0;
    return
        ((c==-1 && cs==s) ?
         NULL : s );
}

unsigned strlen(s)
    char *s;
{
    static char *p;

    p = s;
    do ; while (*p++);
    return p-s-1;
}

main()
{
    char s[20];
    int v;
    CLS;
    printf("\n%s\n","Please type the title..");
    gets(s);
    printf("\n%s\n","and the size of vertical space");
    scanf(" %d",&v);
    printf("\n%u",v);
    CLS;
}
```

```

cent(s);
vspc(v);
printf("%s", "this is the next text line"
);
}
cent(s)
char *s;
{
int x, j;
j=strlen(s);
x=(39-j)/2;
for(j=1; j<=x; j++)
rawout(9);
printf("%s", s);
}
vspc(v)
int v;
{
int j;
for(j=0; j<=v; j++)
rawout(10);
rawout(13);
}

```

The **vspc** and **cent** functions.

FIGURE 8.4

Function **vspc** prints, as the name suggests, a number of empty lines, so spacing your text vertically. The **cent** function will centre a title in its line. As always, you have to be careful of string dimensions. For the **cent** function, it's better to work with character arrays of no more than 40 in length, because you wouldn't want to have two-line titles centred. In this example, the dimensions of **s** have been limited to 20.

The **vspc** function takes as its parameter the number of lines that you want to space vertically, and a loop counts out an equal number of **rawout(10)** statements. These have to be followed by a single **rawout(13)** to place the cursor to the start of the line, otherwise it will be left wherever the printing of the title left it. The **cent** function takes a title, a string **s**. This has been defined as an array of up to 20 characters, and function **strlen** has been used to measure the length of your title and assign it to integer **j**. The centre position is given by using the formula – the figure of 39 has been used rather than 40 (characters in a line), because the left margin on the monitor makes a centered title look too far over to the right when using 40. The action of **rawout(9)** is to tab the cursor to the right, and since this is done in a loop the correct horizontal position is reached.

Left, right and middle.

The BASIC functions LEFT\$ and RIGHT\$ can be replaced to some extent in HiSoft C by the use of some of the printing functions or **strcpy**. The manual, for example, suggests the use of **printf("%2s",strcpy(s,string,2))** as a replacement for LEFT\$(string,2) and **puts(string+6)** as a replacement for RIGHT\$(string,6). These actions apply to printing only, however, and they don't allow you to do what you can do so easily in BASIC, that is to assign part of a string to another string variable name. The next few functions, then, are aimed at providing this action, and also with providing the equivalent of a MID\$ action.

```
#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *gets(s)
    char *s;
{
    static int    c;
    static char  *cs;

    cs = s;
    while ((c = getchar())
           !=EOF  &&  c!='\n')
        *cs++ = c;
    *cs = 0;
    return
        ((c== -1 && cs==s) ?
         NULL : s );
}
char *strcpy(dest, source)
    char *dest, *source;
{
    static char *result;
    result = dest;
    while (*result++ = *source++);
    return dest;
}
char *left(s,n)
    char *s;
    int n;
{
    char *p;
    p=s;
    while(n)
    {
        p++;n--;
    }
}
```

```

    *p=0;
    return s;
}
char *right(s,n)
char *s;
int n;
{
    while(*s) s++;
    while(n)
    {
        s--;n--;
    }
    return s;
}
main()
{
    char s[40],str[40];
    CLS;
    printf("\nPlease type some text\n");
    gets(s);
    /*can use fgets to restrict size*/
    strcpy(str,s);
    printf("\n%s%s\n","The left(3) is ",left
    (str,3));
    /*s is still original string*/
    strcpy(str,s);
    printf("\n%s%s\n","The right(3) is ",rig
    ht(str,3));
}

```

Functions to provide the left and right string slicing actions.

FIGURE 8.5

Figure 8.5 tackles the **left** and **right** problem. Now these are ‘skeleton’ functions in the sense that they do just the minimum that is required. There is no checking to see if the numbers that you use are sensible, and a standard string length of up to 40 characters has been assumed. What is being done is to use a pointer to shift either the starting address or the finishing address of the string. This means that the string which is supplied **has its pointer temporarily altered by the function**, and the new pointer position returned. In the examples, a string which is an array of up to 40 characters is chopped and the chopped portion is assigned to a string of the same name (with the same pointer). Once again the listing shows the header section as well, and you can see that functions **gets(s)** and **strcpy(dest,source)** have been used. The **strcpy** function is used to ensure that the same string is present for both demonstration runs, because the function **left** puts a zero into the string.

The function **left** requires the parameters **s**, the pointer for a string of up to 40 characters, and the position integer **n**. Pointer **p** is used as the position indicator for the characters in the string, and the action consists of shifting this pointer to the end position of the sliced string, **n** characters on, and then placing a zero following the **n**th character. The function then returns the original string pointer to the start of the string, but since the zero has been inserted after **n** characters the string will now appear to be of this length. The function is used by typing **left** with the correct parameters, either to give a new assignment to the existing string, in an assignment (using **strncpy**) to another string, or as part of a **printf** statement. Note that the function allows no protection from a silly number, like **left("Hi",5)**. In a 'skeleton' function like this protection is omitted, but for a program which might be used by anyone who was unaware of the restriction, you would need to carry out a check based on detecting the end of the string in the **while** loop. The **right** function carries out a rather different action, using only the main pointer **s**. First of all **s** is advanced to the end of the string by a **while** loop, which tests for the zero character at the end. When this position is found the pointer is decremented, along with the counter **n** until **n** reaches zero. This places the pointer to the correct number of characters away from the last (non-zero) character in the string. This time, what is returned is the altered pointer. Like **left(s,n)**, the **right(s,n)** action can be used as part of **printf** or independently to change an assigned string. Once again, there is no protection against reading to the end of the string and then counting back too many characters, probably into another string. As before, if you want protection you will have to add it in the shape of detecting a zero character. A useful alternative form of protection for both types of slicing function is to compare **n** with **strlen(s)** before attempting to slice, and returning with the string unaltered if the specified number would cause problems.

The equivalent of the BASIC MID\$ action is carried out by the MID function in Figure 8.6.

```

#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *gets(s)
    char *s;
{
    static int    c;
    static char *cs;

    cs = s;
    while ((c = getchar())
        !=EOF  &&  c!='\n')
        *cs++ = c;
    *cs = 0;
    return
        ((c==-1 && cs==s) ?
        NULL : s );
}

```

```

char *mid(s,pos,len)
char *s;
int pos,len;
{
    char *p;
    while(pos-1){
        s++;pos--;
    }
    p=s;
    while(len){
        p++;len--;
    }
    *p=0;
    return s;
}
main()
{char s[40];
CLS;
printf("\nPlease type some text\n");
printf("\n(at least 8 characters.)\n");
gets(s);
printf("\n%s%s", "The mid(s,3,4) is ",mid
(s,3,4));
}

```

The **mid** action, corresponding to BASIC MID\$.

FIGURE 8.6

In this function once again, **s** is the pointer to the string which is to be sliced and **pos** is the position of the first character that you want to copy. The integer **len** is the **number** of characters that you want to copy. Some MID\$ instructions in BASIC use this second number as another position number, but in this particular function it is used in the same way as in the MID\$ of Amstrad BASIC. The action is the same type as has been used for **left** and **right**, and it consists of incrementing the string start pointer **s** to the position that is given by **pos**. A new pointer, **p**, is then placed at the same address by using **p=s**, and this pointer is then advanced along the string using **len** as a counter. The zero is then placed at the end of the sliced portion of string, and the new starting position is returned as **s**. Like the previous functions there is no protection here against silly numbers, and it's a good exercise in logic to devise a protection against either the impossible starting position or the impossible length of slice, or both.

Another useful function which is provided on many computers, including the BASIC of the Amstrad machines, is STRING\$. This will make up a string of a number of identical characters, with the number and the character specified. For example, in BASIC, X\$=STRING\$(20,"*") will make X\$ a string of twenty asterisk signs. This is a quite simple action to achieve in 'C'. In addition, most varieties of BASIC have the STR\$ function. This converts a number variable into a string variable; something that is quite often needed in BASIC. 'C', with its very flexible rules about data types, is pretty good about conversions, but mainly in the other direction.

```

#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *gets(s)
    char *s;
{
    static int    c;
    static char *cs;

    cs = s;
    while ((c = getchar())
           !=EOF  &&  c!='\n')
        *cs++ = c;
    *cs = 0;
    return
        ((c== -1 && cs==s) ?
         NULL : s );
}
char *fill(s,n,c)
char *s,c;
int n;
{
    while(n){
        *s++=c;
        n--;}
    *s=0;
    return s;
}
char *str(s,n)
char *s;
int n;
{
    sprintf(s," %d",n);
    return s;
}
main()
{
    char s[40];
    int x;
    CLS;
    x=2345;
    str(s,x);
    printf("\n%s - %s\n","String version is
    ",s);
    fill (s,20,'*');
    printf("\n%s",s);
}

```

How to fill a string with a character, and the conversion from number form to string form.

FIGURE 8.7

Figure 8.7 shows the `STRING$` equivalent, `fill`, and the `str` functions. Note that the dollar sign cannot be used as part of a function name in 'C'. The `fill(s,n,c)` function emulates the `STRING$` action, but you have to remember the method of specifying the character. The parameters that have to be passed are the string pointer (name), the number of characters and the character that is to be used for filling the string. As usual, the function counts the number of characters into the array, using `*s++=c` to put a character `c` into the string at pointer number `s`, and then increment the pointer. By using `n` as a counter, the correct number is put in place. The ending zero is then added by using `*s=0`, and the original string starting address is then returned. Remember that the use of `s` in the function does not affect the string pointer that is passed to the function, though we might affect the string characters by use of the pointer. This function is called by typing `fill`, followed, in brackets, by the string name, the number of characters and the filling character. Note that only one character can be specified in one `fill` instruction, and that the character is surrounded by a *single* quote each side, not a double quote. Remember, 'a' is a character, "a" is a string. The result of `fill` is the string `s` filled with the characters. There is no check on the number that is passed to this routine, so that it would be possible to stop the program by specifying a number which was greater than the dimensioning of the array. Since these are skeleton functions, you can fill in the fleshy bits for yourself.

The integer to string a function is even simpler. The string name and the number are passed to a `sprintf` statement, which carries out the conversion. There is really little point in making a separate function out of this one, and you would normally make direct use of `sprintf`. Converting from a string form of a number into the form of a number variable can also be carried out by a built-in function, `sscanf`. For example, if string `s[]="123"` and `n` is declared as an integer, then using:

```
sscanf(s, "%d",&n);
```

will assign the number 123 to variable `n`. This carries out the action of `VAL` in BASIC.

Concatenation and insertion.

Concatenation means joining two strings together, and it is achieved in most dialects of BASIC by using a `+` sign between strings. There is no similar instruction in HiSoft C, but there is a very useful `strcat` function in the library and we can easily write a skeleton function for the insert action. Insertion in this sense means that one string is added to the other at some intermediate position. You might, for example, insert `DIME` into `SENT` so as to get `SEDIMENT`, though it's hardly likely to be something that you would do very often! Nevertheless, both concatenation and insertion are useful actions at times and they can be provided with the functions of Figure 8.8.

```
~#define CLS rawout(12)
#define NULL 0
#define EOF -1
char *strcat(base, add)
    char *base, *add;
```

```

    {
        static char *dest;

        dest = base;
        while (*dest) ++dest;
        while (*dest++ = *add++);
        return base;
    }
char *ins(s1,s2,n)
char *s1,*s2;
int n;
{
char *q,*p,tmp[20],*t;
p=s1;
while(n--)p++;
q=p;
t=tmp;
while (*p) *t++=*p++;
*t=0;
while(*s2)*q++=*s2++;
t=tmp;
while(*t)*q++=*t++;
return s1;
}
main()
{
    static char s1[20]="sent";
    static char s2[]="dime";
    static char s3[20]="sent";
    static char s4[]="dime";
    CLS;
    printf("\n%s%s\n","strcat gives ",strcat(s1,s2));
    printf("\n%s%s\n","ins gives ",ins(s3,s4,2));
}

```

Concatenation and insertion of strings using functions.

FIGURE 8.8

Looking first at the main program, two sets of strings are declared and initialised. One pair, **s1** and **s2**, is concatenated by using the **strcat** function which has been copied from the library. The other pair, **s3** and **s4**, is subjected to the insertion routine. In this routine, two strings and a number are passed. The number decides how many characters of the first string are allowed before the second string is inserted. In this case, since **n=2**, the result should be **se** from string **s3**, then **dime** from string **s2**, then the remaining **nt** from string **s1**, giving **sediment**.

The **strcat** action is fairly simple. Pointer **dest** is assigned to the start of the first string and the first **while** loop then moves the pointer to the last character in the string, the one preceding the zero. The next **while** loop then transfers characters from the second string on to the end of the first one, overwriting the zero at the end of the first one. This continues until the final zero of the second string has been assigned, and the program then returns the original starting pointer. Since the characters have been changed in the routine, the string that we passed to the routine is now changed, so that we can't use string **s1** in the second part of the main program.

The action of **ins** is slightly more complicated. For that reason, the programming has been left in a simple form rather than compacting it. The first action is to set pointer **p** to the start of the first string. A **while** loop then advances this pointer by the number of places given by the number, **n**. Note that there is no error-trapping here, and you might want to ensure that a silly value of **n** could not corrupt the string. The pointer **q** is then used to hold this position, the position of insertions, and pointer **t** is used in another **while** loop to put the remainder of the first string into another string variable, **tmp**, which is dimensioned to 20 characters. This is another place where you might want to put an error trap to prevent this string from being over-filled. The zero is placed at the end of this string (can you think of a better method having seen **strcat** in action?), and then characters are copied from the second string. Finally the remaining characters from the first string are copied from **tmp**, and the pointer to the first string is returned.

Other string and character functions.

The HiSoft manual illustrates a method of obtaining the **INSTR\$** action of BASIC, and also for the more usual action of obtaining a starting position for a string. The normal action of Locomotive BASIC is to get the position number of one string inside another. For example, using **INSTR(JOHN SMITH,SM)** should give the answer 6 since the 'S' of 'SMITH' is the 6th character.

```
#define CLS rawout(12)
#define NULL 0
#define EOF -1

int strncmp(s1, s2, n)
  char *s1, *s2;
{
  if (!n) return 0;
  while (*s1 == *s2)
  {
    if (! *s1) return 0;
    if (! --n) break;
    ++s1; ++s2;
  }
  return *s1 - *s2;
}
```

```

unsigned strlen(s)
char *s;
{
    static char *p;

    p = s;
    do ; while (*p++);
    return p-s-1;
}

int inst(main,sub)
char *main,*sub;
{
    static int len;
    char *s;
    len=strlen(sub);
    s=main;
    do{
        if(!strcmp(main,sub,len))return(main-s
    );
        }while (++main);
    return 0;
}

main()
{
    static char str[]="teststring";
    static char bit[]="tst";

    printf("\n%s%d","Position is ",inst(str,
bit));
}

```

A modification of the library function for the INSTR action.

FIGURE 8.9

Figure 8.9 shows an alternative version of the library function which will always provide the position number, or zero if the string cannot be found. There is no provision in this routine for starting the search at a given number of characters along the main string, but this is something that you can easily add if you want it. All that has to be added is the integer number, which is then added to the starting pointer of **main**. The action of the routine which is shown depends heavily, like the corresponding HiSoft library routine, on the use of **strlen** and **strcmp**. The length of the small string is obtained as the integer **len**, and the pointer **s** is set to the start of the main string. The **do..while** loop makes a test and continues until the test succeeds or until the end of the main string when the zero character is detected after incrementing the **main** pointer number. The test is that **strcmp(main, sub,len)** is not zero. The **strcmp** function will return a zero unless two strings of the same length **len** are identical. The loop will therefore cycle round, advancing the starting character until **len** characters are identical in the two strings. If this is found, then the return is of **(main-s)**, the difference between the incremented pointer and the original value for **main**. This is the position number for the first character of the small string. The only

safeguard in this routine is that a zero is returned if the end of the main string is found before a match can be found, and this causes a problem if the two strings are identical. The character count in this example starts from zero, so that identical strings give the same result as no match. If this is likely to cause trouble, use a `return(main+1-s)` in the `inst` function.

In addition to the string functions that we have looked at in detail, there are several others which are sometimes useful. Function `strchr` will give a pointer to the first place in a string where a given character occurs. The manual illustrates the main use for this function which is to check that a character belongs to a set, such as the letters in a Roman number, the numbers of the weekdays, or whatever. You can also see from the example that this is one of the actions which Pascal does rather more neatly than 'C'! There is also a similar function `strrchr`, which returns the *last* position of character `c` in the string. This can find, for example, the surname in a name that consists of more than two parts by getting the last space in the name. Three other functions are rather more exotic. Function `strpbrk` compares two strings, and finds the first place in 'string1' where any character from 'string2' occurs. This one is of use if you are looking for the occurrence of particular letters for some reason. The `strspn` and `strcspn` functions both look at the start of a string, `s1`, and compare it with a string `s2`. If the first few letters of `s1` contain letters from `s2`, then `strspn` returns the number of letters. The function `strcspn`, by contrast, returns as many letters at the start of `s1` as are *not* in `s2`.

In addition to these string functions there are many useful functions which operate on characters, one by one. The most useful of these character functions are built-in, and they consist mainly of character analysing actions. Looking at the built-in functions to start with, you can use `isalpha` to decide if a character is a letter, upper or lower case, and `isdigit` to decide if a character is a digit. The other three built-in tests are `islower` to test for lower-case, `isupper` to test for upper-case, and `isspace` to test for 'whitespace'. Remember that a 'whitespace' character can be the spacebar character, the newline, or the tab character. Also built in are two character changing functions, `tolower` and `toupper`, which change the case of a character. There are also several tests which are not built in: for alphanumeric characters (letters or digits); for ASCII codes; for control characters; for graphics characters; for printing characters; or for hex digits. All of these have their uses particularly if you want to *parse* a phrase, of which a little more later. Meantime, Figure 8.10 illustrates some of these tests and conversions in action.

```
#define CLS rawout(12)
#define NULL 0
#define EOF -1
unsigned strlen(s)
    char *s;
{
    static char *p;
    p = s;
    do ; while (*p++);
    return p-s-1;
}
char *cap(name)
char *name;
```



```

{
char *p;
int j,len;
p=name;
len=strlen(name);
while(*p)
{
if (p==name) *p=toupper(*p);
if (isspace(*p))* (p+1)=toupper (*(p+1));
p++;
}
return name;
}
main()
{
static char name[]="john smith williams"
;
CLS;
printf ("\n%s%s\n", "Name is ",name);
printf ("\n%s%s\n", "Name is now ",cap(name));
}

```

Illustrating the character tests and conversion functions.

FIGURE 8.10

The aim here is to analyse the letters of a name and convert correctly to upper case. The principle is that the first character ought to be in upper case, and any character which follows a space should be in upper case. Perhaps you might like to try altering this routine so that it could deal with names that contained a hyphen? The important points, however, are to see the tests in action, and how the changing functions are used.

The main intended use of all these tests, however, is in writing interpreters and compilers. The 'C' language was developed just for such purposes, and this accounts for the number of functions which analyse and work with characters. One of the important features of a language interpreter or compiler is 'parsing'. This (for the sake of readers younger than 40) means separating the parts of a sentence to explain what each part does. In a BASIC statement such as PRINT C, for example, PRINT is the word that describes the action, and C is the name of a number variable. A parser function can deal with a piece of text like this in at least two ways. One way is to go through the letters, looking for the space and then splitting the phrase into its two parts. If this is done, then a statement such as PRINTC will be rejected because the space is missing. The alternative is to use the **strspn** type of action to detect the PRINT, and then separate the 'C' by using the number from **strspn**. This is a more common method, and in most BASIC interpreters a code number (the 'token') is then substituted for the action word. In this book, there simply isn't space to start considering the fascinating work of writing a compiler or interpreter. If you are interested, I can thoroughly recommend the book 'Writing Interactive Compilers and Interpreters', by P.J. Brown (John Wiley). Apart from anything else, it's a thoroughly readable, interesting and useful book for anyone with serious programming interests.

Chapter 9

Graphics, sound and BASIC conversion.

The excellent graphics and sound capabilities of the Amstrad micro would be wasted if it were not possible to make use of them from HiSoft C. This is done by making use of a library of functions which carry out the graphics and sound actions. In addition there is another library which carries out actions such as setting function keys, which are available in the BASIC of the Amstrad machines. Since the use of these graphics and sound instructions in BASIC is well documented elsewhere, I shall not take up space in this book with them. In general, all of the normal instructions that Locomotive BASIC uses for graphics can be implemented in 'C' with a few slight changes in syntax and one major enhancement. The SOUND statement of BASIC, however, has been abandoned in favour of a scheme which is closer to the methods used on the IBM and MSX computers (and Dragon, if anyone remembers!).

As an illustration, we'll look at a simple piece of graphics executed by 'translating' a BASIC routine.

```
10 MODE 1
20 CLG
30 FOR N=1 TO 50
40 MOVE 320,200
50 DRAW RND(1)*639,RND(1)*399,RND(1)*4
60 NEXT
```

A graphics routine in BASIC.

FIGURE 9.1(a)

```

#include stdio.h
/*Simple graphics*/
main()
{
int n;
rawout(4);rawout(1);
/*set mode 1*/
G_clear_window();
/*CLG*/
for (n=1;n<=50;n++)
{
G_move_absolute(320,200);
/*MOVE(320,200)*/
G_set_pen(rand()%4);
/*random colour*/
G_line_absolute(rand()%399,rand()%399);
/*MOVE*/
}
rawin();
/*hold pattern*/
}
#include ?basic2.lib?
#include ?stdio.lib?

```

The conversion into 'C', using the **basic2.lib** routines.

FIGURE 9.1(b)

The BASIC version is shown in Figure 9.1(a), the 'C' version in 9.1(b). The BASIC statements have been 'translated' into 'C' functions and statements, using the HiSoft Manual list of equivalents. Because these call for functions from the **stdio.lib** and **basic2.lib**, the header **stdio.h** must be placed first in the listing and the other library names put at the end so that the compiler can select functions in the usual way. Most of the translations are fairly direct, but some changes are needed to random numbering. The random number function of Locomotive BASIC gives a random number which is always less than 1. The **rand** function in the **stdio.lib** generates a number which is an integer, and which is not so random as the RND of BASIC. You can see this when you run the program, because the pattern is always the same. Even if you try to alter the randomness by using **srand(n)** at the start of the loop, you'll find that the patterns repeat. This is a consequence of not being able to use floating-point numbers, which prevent the ROM routines for random number from being effectively used. You can, however, make use of other random number routines. In place of the RND(1)*399 type of routine that is used in BASIC, the 'C' version uses **rand()%399** to create a 'random' integer and find its modulus with 399 – in other words, to find a remainder between 1 and 398.

Any program in BASIC which deals with graphics can make use of a 'C' translation of this type. It's much more interesting, however, to make use of the new routines that 'C' permits.

The main new graphics routine is **draw()**, and its action will certainly not be familiar to Amstrad owners though it is well known to users of the MSX machines. Knowing the action is one thing, knowing the syntax is another. Like the function which is available for sound, the **draw** function uses position numbers which don't bear much similarity to the numbers that are used in the BASIC of the Amstrad machines. In addition, the numbers that are used must be expressed in a way that is certainly not familiar to many programmers nowadays, – octal. There's more about octal scales in Appendix A, but to introduce the **draw** action, take a look at the listing of Figure 9.2.

```
#include <stdio.h>
#define CLG inline(0xcd,0xbbdb);
main()
{
char *string;
CLG;
string="0 m\0\0\217\1 1\177\2\0\0\0";
draw(string);
rawin();
}
#include "basic2.lib"
```

An illustration of the **draw** function in the **basic2.lib** library. Note that the numbers are in the octal scale.

FIGURE 9.2

This defines a string:

```
"0 m\0\0\217\1 1\177\2\0\0\0"
```

which is then used in a **draw** function. The result of all this effort is a horizontal line across the screen, half way up and extending all the way across. How does the string cause this effect?

The answer is that the letters act as commands for graphics actions. The complication is that each letter has to be followed by numbers, anything from zero to four numbers, and in octal. The three commands that are illustrated are **0** (the zero byte), **1** (ell) and **m**. The **0** command causes the graphics cursor to go to the bottom left-hand corner, the (0,0) position. This command needs no numbers following it. The **1** command moves the cursor, but without leaving any trace. It needs four numbers following it. The **m** command letter means move, and it draws a line. It also needs four numbers following it. The range of numbers is the same as the standard Amstrad BASIC range. The range of X numbers is 0 to 639, the range of Y numbers is from 0 to 399. These are denary numbers, and the ones that follow the commands are not. The conversion is by no means simple, because each number has first to be converted into two bytes, in low-high order, and then into octal.

The conversion is carried out in stages. First of all, if the number is negative, it has to be subtracted from 65536. If the position number that you want to convert is then less than 256, then it fills only one byte of memory with zero in the upper byte. A number such as 190, for example, can be written as 190,0 – meaning a low byte of 190, high byte of zero. If the number is greater than 255 but less than 512, then its high byte is 1 and its low byte is obtained by subtracting 256. For example, the number 320 has a high byte of 1, and a low byte of $320-256=64$. It would be written in two byte form as 64,1. If the number is greater than 512, it has a high byte of 2, and the low byte is the number less 512. For example, 600 has a high byte of 2, and a low byte equal to $600-512=88$, and would be written as 88,2. If the position number started off as negative, then subtracting from 65536 produces a large number and it's easier to find the high/low bytes using the alternative procedure. The high byte is $\text{number}/256$ (integer division) and the low byte is $\text{number}\%256$ (the modulus). Either procedure gets numbers into two-byte form. The next step is to get them into octal form. Take a number such as 88, divide it by 8, which gives 11 and no remainder. Write down the remainder, which is 0. Now divide the 11 by 8, getting 1 and a remainder of 3. Write down the 1, then the 3, then the final zero, and you have 130, the octal version of 88 denary. If you find this a nuisance, then you have two options. One is to buy an octal calculator, such as the TI[®] LCD Programmer, the other is to make use of the program in Figure 9.3.

```

/*Octal bytes*/
unsigned oct(n)
unsigned n;
{
    unsigned y,x,b;
    y=x=0;b=1;
    do{
        if(y!=0)n=y;
        x+=b*(n%8);
        b*=10;
        y=n/8;
    }while(y>=8);
    x+=b*y;
    return x;
}
unsigned negit(inp)
char *inp;
{
    unsigned j;
    inp+=1;
    sscanf(inp," %d",&j);
    return (65536-j);
}
main()
{
    unsigned j,lo,hi;
    char inp[5];
    do{

```

```

printf("\nDenary number - \n");
scanf(" %s",inp);
if (*inp=='-') j=negit(inp);
else sscanf(inp," %d",&j);
lo=j%256;
hi=j/256;
printf("\n %s \\%d \\%d \n","Octals ar
e ",oct(lo),oct(hi));
}while(j!=0);
}

```

A program for performing conversions from denary to octal, positive or negative.

FIGURE 9.3

This carries out the conversions for any number in the approved range of 0 to 640, positive or negative – you enter the number when asked, and you get a printout of the two bytes in the correct order. The bytes are written with the backslash to ensure that each one is stored as one character, rather than as an integer.

With the octal bother out of the way, we can give our attention to the graphics string commands. The **m**ove and **l**ine commands each cause the cursor to move to a new position which is 'x' points to the right of the old one, and 'y' points up. If you want to move left or down, you will have to put a negative sign into the denary number, and then convert. The program of Figure 9.3 will cope with negative numbers. Each number will be in two-byte form, even if the number is a small one, and must be in octal, using the backslash. The only absolute position command is 0, which means that the cursor moves to the origin, point 0,0, the bottom left-hand corner. The **m** command can then be used to position the cursor, the **p** command to position the cursor and light a point and the **l** command to draw a line. The string of commands must end with a final '/0' so that the compiler can recognise the end. Suppose we wanted to draw a square somewhere around the centre of the screen. Figure 9.4 shows the routine which would do this.

```

#include <stdio.h>
#define CLG inline(0xcd,0xbbdb);
main()
{
char *string;
CLG;
string="0m\35\1\257\0 1\0\0\62\0 1\62\0\
0\0 1\0\0\316\377 1\316\377\0\0\0";
draw(string);
rawin();
}
#include ?basic2.lib?

```

A square drawing routine.

FIGURE 9.4

The CLG command has been put in with a `#define`, though it could just as easily have been provided from the `basic2.lib`, since this library has to be used in any case. If, incidentally, you find that you get an error message while the `stdio.h` is being read, this is always due to memory corruption and you will have to switch off and completely reload.

The string is the one that will draw the square. The starting zero puts the cursor to the bottom left hand corner of the screen and the following four numbers of the `m` command place it at the bottom left hand corner of the square position, which is $x=285$, $y=175$ in denary numbers. These translate into the octal set `\35\1\257\0`. From this starting point, the sides of the square are then drawn. The first step is to draw a vertical side by specifying a y -change of 50 and an x -change of 0. This makes use of the octal set `\0\0\62\0`, with the `l` command used for drawing the line this time, not just moving the cursor. The next line is the top of the square, a movement of 50 steps in the positive y -direction. This uses the octal sequence `\62\0\0\0`, and from this point, the movements will need negative co-ordinates. The movement of -50 in the y direction is programmed using octal `\0\0\316\377`, and the last movement of -50 in the x -direction uses `\316\377\0\0`, and the string ends with another `\0`. When this runs, you'll see the square drawn very quickly. The `rawin()` line ensures that the appearance of the square is not disturbed by the 'Type y to run:' message until a key is pressed.

The use of a string of commands like this is rather restricting, because you can type only about two lines of a string using the keyboard. You can, however, use a set of `draw` statements in sequence, because each one can take over where the other left off. One particularly useful way of doing this is to arrange the strings into arrays, and then use a loop to `draw` the array strings.

```
#include stdio.h
#define CLG inline(0xcd,0xbbdb);
main()
{
  int n;
  char *string[2];
  CLG;
  string[0]="0m\35\1\257\0 1\0\0\62\0 1\62
\0\0\0\0";
  string[1]="1\0\0\316\377 1\316\377\0\0\
0";
  for (n=0;n<=1;n++)
  draw(string[n]);
  rawin();
}
#include ?basic2.lib?
```

How an array of `draw` strings can be used.

FIGURE 9.5

Figure 9.5 illustrates this with a simple example, the square pattern drawn as two sections with an array of two pointers. This allows you to make as much use as you need of the **draw** strings and also makes it possible to test drawings step by step, with one step assigned to each pointer in a pointer array.

```
#include <stdio.h>
#define CLG inline(0xcd,0xbbdb);
main()
{
  int n;
  char *string[2];
  draw("cb\0");
  CLG;
  string[0]="0m\35\1\257\0 cf\1 1\0\0\62\0
  1\62\0\0\0\0";
  string[1]="cf\3 1\0\0\316\377 \21 1\316
  \377\0\0\0";
  for (n=0;n<=1;n++)
  draw(string[n]);
  rawin();
}
#include ?basic2.lib?
```

Adding colour instructions to a **draw** string.

FIGURE 9.6

Figure 9.6 illustrates this by using the separated strings to add colour instructions. The colour command letter is **c** and it is followed by the letter **\‘f** or **\‘b**, meaning foreground colour or background colour respectively. These colours are taken from the normal Amstrad INK range. For Mode 1, this consists of four colours using numbers 0 to 3. You can, of course, change the colour set by using the **ink()** function in the **basic1.lib** set of routines. The usual rules of PEN and PAPER colours are followed. If you want to change the background colour you need to follow it with a CLG instruction, so that a separate **draw** string is useful. In Figure 9.6 this has been done, using the conventional INK(0), dark blue, as background, but with two different foreground colours. Note the syntax for the colour changes, using **cf \ 1** and **cf \ 3**.

More spectacular changes can be achieved by using the scale-changing features of command **s**. Command **s** is followed by the usual four numbers, two for each dimension, and it causes the scale of the drawing to be changed. The original scale is represented by a factor of 4, so that using the string **“s \ 4 \ 0 \ 10 \ 0”** will make all of the x-dimensions normal, as specified, and all of the y-dimensions double scale.

```

#include <stdio.h>
#define CLG inline(0xcd,0xbbdb);
main()
{
int n;
char *string[2];
draw("cb\0");
CLG;
draw("s\4\0\10\0");
string[0]="0m\35\1\127\0 cf\1 1\0\0\62\0
1\62\0\0\0\0";
string[1]="cf\3 1\0\0\316\377 \21 1\316
\377\0\0\0";
for (n=0;n<=1;n++)
draw(string[n]);
rawin();
}
#include ?basic2.lib?

```

Using the scale-change command letter s.

FIGURE 9.7

In Figure 9.7, you can see that this causes the drawing to be stretched into a rectangle. Note that the starting point has had to be altered to allow for the change in the y-dimensions. The scale changing can be examined in more detail in Figure 9.8.

```

#include <stdio.h>
#define CLG inline(0xcd,0xbbdb);
main()
{
int n;
draw("cb\0");
CLG;
pattern();
draw("s\6\0\6\0\0");
pattern();
draw("s\10\0\10\0\0");
pattern();
draw("s\12\0\12\0\0");
pattern();
rawin();
}
#include ?basic2.lib?
pattern()

```

```

{
    char *string[2];
    int n;
    string[0]="0 cf\1 1\0\0\62\0 1\62\0\0\0\
0";
    string[1]="cf\3 1\0\0\316\377 \21 1\316
\377\0\0\0";
    for (n=0;n<=1;n++)
    draw(string[n]);
}

```

Illustrating the scale sizes, which are applied relative to the previous drawing.

FIGURE 9.8

In this example the drawing has been put into a function, and the scale strings are used in the **main**. If you look at the sizes of the squares which are drawn in this routine, you'll see that they are by no means proportional to the (octal) numbers which follow the ratio 4:6:8:10 in denary. This is because each scale string *operates on the previous string that was drawn*. For example, supposed that we used scales of 4,5,6 and 7 in sequence. If the square drawn with the scale of 4 (normal 1:1) had a side of 50 points, then the square drawn with scale 5 would have $50 * 5/4 = 62$ points (no fractions allowed). This, however, would be the size of the square for the next action, so that the scale of 6 would produce a square of side $62 * 6/4 = 93$ points, and so on. To prove this point, try making each scale number equal to 5 and you'll see a group of squares drawn with equal increments in size. This is a point which is not made clear in the manual, and only emerges when you study the routines in the library that are used for the graphics functions. It's a good illustration, in fact, of the action of static integers in a function.

Finally in this section, the **r** command letter will cause a pattern to be drawn rotated by one right angle in the clockwise direction. Once again, this is a command which has to be used with some understanding. The effect of **r** is to rotate everything that follows it. If, for example, you follow the **r** command with a move from 0,0 (bottom left hand corner) to the centre of the screen, then the rotation will attempt to move from the corner to some point off the screen. Any movements like this *must* be made before the **r** command is used.

```

#include stdio.h
#define CLG inline(0xcd,0xbbdb);
main()
{
    int n;
    draw("cb\0");
    CLG;
    draw("0 s\4\0\10\0 m\35\1\127\0\0");
    pattern();
    draw("r\0");
    pattern();
    draw("r\0");
    pattern();
}

```

```

draw("r\0");
pattern();
rawin();
}
#include "basic2.lib"
pattern()
{
char *string[2];
int n;
string[0]="cf\1 1\0\0\62\0 1\62\0\0\0\0"
;
string[1]="cf\3 1\0\0\316\377 \21 1\316
\377\0\0\0";
for (n=0;n<=1;n++)
draw(string[n]);
}

```

Illustrating the rotation of a drawing.

FIGURE 9.9







The program in Figure 9.9 shows the old square-drawing program altered to suit. The scaling and the movement of the cursor are all done before the remainder of the routine is called as a function. This shape is then rotated three times, forming a pattern of rectangles. Notice that the rotation is always around the first point in the pattern, the one which formed the bottom left hand corner of the original pattern.

Sounds unlimited.

In the space of this book a really full explanation of the sound commands of the Amstrad machines is not appropriate, and I have to refer you to specialised books modestly naming no names. The methods by which sound can be controlled using HiSoft C is, however, rather different from the methods of Locomotive BASIC, and for that reason requires rather more background information than would be needed otherwise. The Amstrad BASIC SOUND commands require all of the instructions to be in number form. If you read music, or can work with sheet music, this is the last thing that you want. The ideal method of programming music would be to work with the named notes of music – and this is what is done when you use the **play()** routine in the HiSoft C **basic1.lib** library. It might appear to be the obvious thing to do, but very few computer languages do it!

If you have no experience of music, however, this may seem rather puzzling to you. How do we go about writing down music? For each note we have to specify what the note is (its pitch), how loud it must be and for how long it is to be played. In written music this is done by using a type of chart for the pitch, and different shapes of markings (notes) for the duration. Loudness is indicated by using letters such as **f** (loud) and **p** (soft). More than one letter can be used, so that **fff** means very loud, and **ppp** means very soft. Each sound is






indicated by a note, a shape on the chart, and the shape of the note gives some information about the duration of the note. In addition to this, each piece of music will start with some advice about the speed at which the notes are to be played. One of these methods is a metronome reading. The metronome is a gadget which ticks at regular intervals, and the metronome reading for a piece of music is the number of metronome ticks per minute. A more ancient way of indicating speed is the use of Italian words like *allegro* (fast), *lento* (slow) and so on. What these speed settings decide is how many unit notes will be played in a minute. The unit note is the crotchet, so if a piece of music is marked at a metronome speed of 60 (pretty slow), then there will be 60 crotchets played per minute. The durations of all the other notes are decided in comparison to this unit, the crotchet. A minim sounds for twice as long as the crotchet, a semibreve for twice as long as a minim which is four times as long as the time of a crotchet. The quaver sounds for only half the time of the crotchet. A semiquaver sounds for only half the time of a quaver, which is quarter of the time of a crotchet. The crotchets and other timed notes are indicated by the shapes of the written notes, as Figure 9.10 shows.

Symbol	Time (relative to crotchet)	Name
	$\frac{1}{8}$	Demisemiquaver
	$\frac{1}{4}$	Semiquaver
	$\frac{1}{2}$	Quaver
	1	Crotchet
	2	Minim
	4	Semibreve

How the shapes of the note symbols are used in written music.

FIGURE 9.10

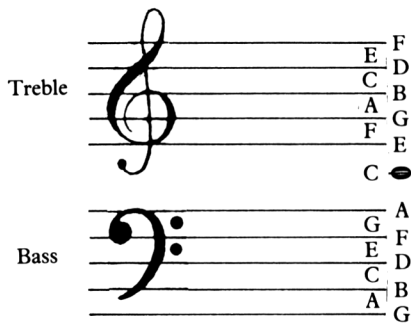
In addition symbols are used to indicate silences in the music, and these are based on the same idea of a unit duration of silence and others which are twice, four times, half, or quarter. These silence marks are shown in Figure 9.11

Rest symbol	Time (relative to crotchet)
	$\frac{1}{4}$
	$\frac{1}{2}$
	1
	2
	4

Silence marks in music.

FIGURE 9.11

The pitch of a note is indicated in written music by placing it on to a kind of musical map which is called the 'stave'(Figure 9.12).



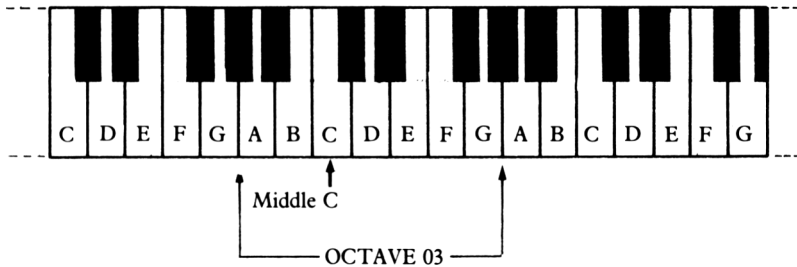
The stave and how notes are placed on it.

FIGURE 9.12

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

The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes called the 'scale of C Major'. The scale starts on a note that is called Middle C and ends on a note that is also called C, but which is the eighth note above Middle C. A group of eight notes like this is called an 'octave', so that the note you end with in this scale is the C which is one octave above Middle C. Because music (in the Western hemisphere, at least) is based on this group of eight notes, we use only the first seven letters of the alphabet in naming the notes. Why 7?

Well, the eighth note is the end of one octave and the start of the next, so it bears the same name. The scientific basis of all this is that if you take Middle C and find the frequency of the sound of this note, then the C which is the next octave above Middle C has precisely double the frequency value of Middle C. The C below Middle C has half the frequency of Middle C, and so on. That's why the ancient Greeks always thought that music was a branch of mathematics.



Arrangement of keys on a piano keyboard.

FIGURE 9.13

The appearance of these keys on the piano keyboard is illustrated in Figure 9.13. Middle C is, logically enough, at the centre of the keyboard, and we move right for higher notes or left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a 'tone' and the smaller spacing is called a 'semitone'. Each scale will contain two semitones. On written music, Middle C appears midway between the treble and bass staves.

The key instruction for playing music with HiSoft C is the **play()** function. Like **draw()**, **play** has to be followed by a string name and a channel number. The string then contains all the information that is needed to produce the music. The channel number is used in exactly the same way as it is used in Locomotive BASIC, with numbers 1 to 7 for channels and the higher numbers used for synchronisation, holding a channel, or clearing sound buffers. The notes are specified simply by their names, as used in music. These are the letters A to G, with upper-case letters used, and we also use the signs # and b. The # sign means a semitone *higher* than the note indicated by the letter, so that A# is a semitone above A – the note a musician would call 'A-sharp'. Similarly, Ab would mean a semitone *below* A, or 'A-flat'. In addition to the letter names of the notes, we can use other control letters to indicate the octave, volume, tempo, amplitude envelope, pitch envelope and noise. The octave letter is (upper-case) **O**, and it has to be followed by a number whose range is 0 to 8. If you don't specify any 'O' value, the computer will set itself to **O3**, which is the octave that contains Middle-C. The actual range of an octave for the purposes of the **play()** function is from A to G, rather than the usual C to B range that is used with computers. **O0** means the lowest range of computer notes; **O8** gives the highest. This means that the computer can play nine octaves of notes, which is more than the range of any ordinary musical instrument. The volume control letter, **V**, can be followed by a number whose range is 0 to 15. This lets us make music whose volume can change during the playing of the music. As you might expect **V0** gives the lowest volume, **V15** the greatest.

The computer sets to **V12** if you don't specify anything different. We can, of course, still set the volume control on the computer itself to suit our own tastes. The duration of a note is controlled by a number in the range 1 to 40 octal, and the default setting is 4. This duration number, if needed, is placed following the note name. The duration of each note is mainly set by the tempo of the music, and the number following the note will be used mainly to change specific notes, not with each and every note. The tempo is set by using the **T** command with a number in the range 1 to 377 octal.

Time now for some illustrations. This involves some extra work if you are using the CPC464 model. Start by clearing out any program text, and then loading in the **basic1.lib** text file. When you list this, then, at or near to line 4370 you'll see the statement:

```
reg_bc=0x80FF; /* asynchronous, all RAM*/
```

which is intended for the later CPC664 (of the brief life) and CPC6128 machines. If you are using the CPC464, the statement has to be changed to 0x02FF to correct a problem in the way that the CPC464 handles these routines.

We can then start with Figure 9.14.

```
#include <stdio.h>
main()
{
  setup_sound();
  play("T\74 O\3 C.D.E.F.G.A'.B'.C'.",2);
}
#include "basic1.lib"
```

A simple example of **play** in action. The function **setup_sound** must be called before any **play** action.

FIGURE 9.14

This calls the function **setup_sound**, which prepares the routines for handling the **play** actions. The string is then defined and played in one action by using the **play(string, channel)** form. The notes consist of the scale that starts at Middle C. How do I know? Well, Middle C in the HiSoft C pattern is the third note in octave number 3, so by omitting an octave command (which gives the effect of **O3**) and specifying **C**, we get Middle-C. The other notes have been written in sequence, but when we get to the note **A**, we are starting a new octave. Unless you force the octave number higher, the **A** below **C** will be played. We could use **O4** for this purpose, but a more convenient method is to use the apostrophe sign. The apostrophe following the note will raise the pitch of that note by one octave – you need an apostrophe for each note that is altered in this way. Placing an apostrophe **preceding** a note will lower its octave by one. The scale uses the default value of volume and a speed (tempo) of 74 octal, 60 denary. This corresponds to 60 notes per minute, one per second, which is a nice slow pace. This also is a default setting. Each note in the string is followed by a full-stop, which marks the note as one to be played. The computer will ignore any note which is not followed by this full-stop. When you run this the 'Press y to run:' message appears almost at once, and if you press any key before the music has finished you will stop the sound. Pressing the **y** key will always re-start the scale.

The scale of **C** is a simple one, but it's a good piece of music to illustrate a few of the things that can be done with this **play** command. Try Figure 9.15 now, to see what we can do with the volume command, **V**, and the length numbers.

```
#include <stdio.h>
main()
{
char *music;
setup_sound();
music="T\170 V\2 C.D.E.F. V\7 G\62.A'.B'
. V\17 C\1.";
play(music,2);
}
#include "basic1.lib"
```

Using the **V** command letter to change volume, and using note duration numbers.

FIGURE 9.15

In the example, we have used the ordinary tempo, 120 denary (170 octal), but changed the volume and length of note settings. The reason for having separate tempo and length controls is that you can get the tune sounding right by using the length number to select the length of notes which are not crochets, and then use **T** right at the start to set whatever tempo you like. If you want to speed things up, use a low value for **T**, if you want a funeral march, use a high value. You can even write the string without a **T**, and then add it in by an earlier command like:

```
play("T170");
```

Another useful part of the string of notes is the **W** command letter. This means 'wait', and is used to provide a pause between notes. The pause duration can be controlled in exactly the same way as the duration of a note, by using an octal duration number in the range 1 to 40 following the **W** and separated by a backslash. The full stop must follow the **W** command and its duration number just as if it were a note.

```
#include <stdio.h>
main()
{
char *music[2];
int n;
setup_sound();
music[0]="T\300 G.D\4 A.Bb.W\2.A.D. 'G.W\
2.";
```

```

music[1]="A.D.C.Bb\2.A\2.Bb\2.C\2.Bb.w\4
.A\10.";
for (n=0;n<=1;n++)
play(music[n],2);
}
#include "basic1.lib"

```



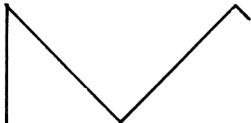
More music, with the **W** command letter used for silences.

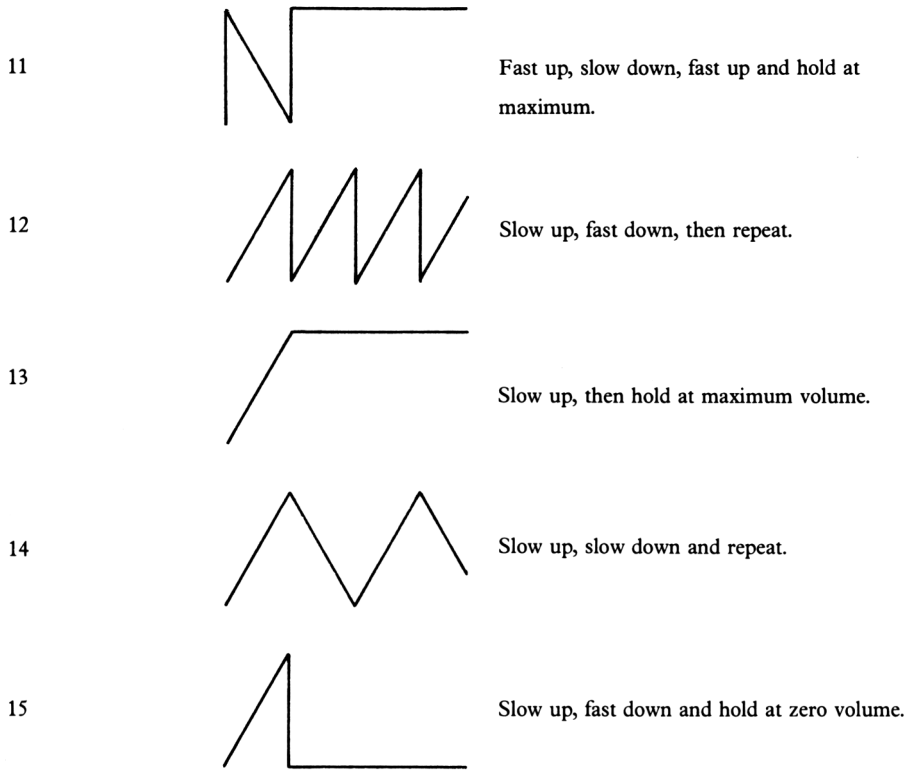
FIGURE 9.16

Figure 9.16 illustrates the use of the wait commands in a piece of music that also contains flattened notes, using the 'b' sign. Unless you are a very experienced musician, you will need to use sheet music to provide the notes for you. If you can read sheet music, it's an easy matter to translate it into the strings that are needed for the **play** function.

Using envelopes.

If you have used envelopes along with the BASIC of the Amstrad machines, you will not find it too difficult to adapt to the use of envelopes with the **play()** function. If you use an early model of the CPC464, however, you may not be aware from the manual of the variety of envelopes that can be created. These are described in detail in a book of mine, *Music & Sound on the Amstrad CPC464*. In this section, we'll look very briefly at the methods by which envelopes can be specified and used with the **play** function. The Amstrad manuals concentrate on 'software envelopes', meaning that you have to design the shape of each envelope for yourself. It's never easy to do this, and one alternative is to make use of the set of standard shapes which are built into the hardware. These are illustrated in Figure 9.17, along with the values of reference numbers which produce them.

Number	Shape	Description
8		Fast up, slow down and repeat.
9		Fast up, slow down, then hold at zero volume.
10		Fast up, slow down, then repeated slow up, slow down.



The hardware envelope shapes of the Amstrad computers.

FIGURE 9.17

For producing a sound with a hardware envelope, then, you need to set up the number of sections (1 to 5), and for each section the values of envelope number and envelope period. Both the number of sections and the envelope number will be a single byte (a character number), one in the range 1 to 5, the other in the range 8 to 15, and the period is an unsigned number in the range 1 to 65535. These numbers can be set up in a form suitable for reading by putting them into a structure. The function that creates the envelope can then be passed the address of this structure, so that the numbers can be read in order.

```
#include <stdio.h>
struct hard {
    char nr;
    char shape;
    unsigned period;
} section;
main()
```

```

{
int j,n,k;
char w;
unsigned x;
setup_sound();
section.nr=1;
for (w=8;w<=15;w++)
{
printf("\n%s %d","Envelope No.",w);
x=1;
for (j=1;j<=4;j++)
{
x*=10;
printf("\n%s %d","Period",x);
section.shape=w+128;
/*must have bit 7 set*/
section.period=x;
S_ampl_envelope(1,&section);
play("Y\1 C.",2);
for (k=0;k<=10000;k++);
}
}
}
#include ?basic1.lib?
#include ?stdio.lib?

```

An illustration of how to set up hardware envelopes, and how they sound.

FIGURE 9.18

Figure 9.18 illustrates how hardware envelopes are set-up, and how they sound. Various values of period are used for the full range of envelope numbers, and the screen prints up the values as you hear the sounds. Several points of detail are important. One is that the envelope must be numbered, both in the call to **S__ampl__envelope** and in **Y** command of the **play** string. The principle of the envelope statement is that the number of sections is set up, and each section requires three bytes of data. For a hardware envelope, each of these sections consists of a shape number and a two-byte period. The shape number consists of the envelope shape code, number 8 to 15 inclusive, *with 128 added to it*. The addition of 128 is vital, because this sets bit 7 of the byte, and is the way that the machine-code routines of the Amstrad machines recognise the difference between a hardware envelope and a software one. There is no need to set duration or volume to zero, as you have to when you use the BASIC sound statements with an envelope. If you do, in fact, the program will hang up with no sound and no loop running, but it can be recovered by using the (ESC) key. The duration number in the **play** string will control the overall time for which the note is sounded.

The program runs slowly because of the time delays that are built in. The time delays are essential, however. The reason is that the sound generator is a little computer in its own right, and if you issue it with a **play()** function it gets on with it independently. If you do not use a time delay in this program then you can find the screen displaying values of envelope and period which are well ahead of what the loudspeaker is playing. This is because the display is fast, but the music has been forced to play at a slower pace. This can be very useful, because it means that if you mix music with other computing actions, you will not be held up while the music plays. You will find that the **play** function of HiSoft C allows many more notes to be put into a queue than the SOUND command of Locomotive BASIC. You will find, as you run Figure 9.18, that several of the envelope values sound pretty much the same. There are in theory only eight different wave shapes, and not all of these can be easily distinguished unless you have a good ear for sound. In particular, if you use short values of period, you will not hear the effect of the 'repeater' notes that you get with envelope number values of 8, 10, 12 and 14. You will find, in fact, that for a lot of notes you can hear only a few main types.

If you haven't used hardware envelopes from BASIC, you might be confused by the 'period number'. The name suggests that its value might set the total time of each envelope. In fact, it sets the time for each step when the waveform amplitude is changing. If you consider envelope shape 13 as an example, it consists of the sloping part, the **ramp** and a steady part. The ramp consists of sixteen steps of amplitude (from 0 to 15), and the period number sets the time between each step. The Amstrad hardware manual states that the step time is 128 microseconds. My own timing indicates that this may be a misprint, and that the time per step should be 12.8 microseconds, corresponding roughly to a period number of 5000 for a step time of one second. Step times as long as this are useful only for special effects. Another point to note as regards timing is that when you use machine code, the firmware places a two-second delay following any hardware envelope. If you want to use hardware envelopes for music then each envelope should use its hardware section(s), followed by a software section of silence which will then set the timing for the overall envelope. The timing in 'C', however, is still controlled by the duration number which is part of the note specification.

```
#include <stdio.h>
typedef struct{
    char shape;
    unsigned period;
}hard;
struct{
    char sect;
    hard section[2];
}soundit;
main()
{
    setup_sound();
    soundit.sect=2;
    soundit.section[0].shape=137;
    soundit.section[0].period=1000;
```

```

soundit.section[1].shape=1;
soundit.section[1].period=10240;
S_ampl_envelope(1,&soundit);
play("Y\1 C\1.D\1.E\1.F\1.G\1.A\1.B\1.
C\1.",2);
}
#include ?basic1.lib?
#include ?stdio.lib?

```

Mixing a hardware section and a software section.

FIGURE 9.19

Figure 9.19 shows how such an envelope can be constructed, using one hardware section and one silence section. A more elaborate structure is needed this time because we must have one structure to contain the number of sections and the section description array, then another structure to contain the parts of each section. The illustration in the manual (under the description of **S_ampl_envelope**) shows how to allow for a mixture of hardware and software envelopes by using a **union**, a topic that will be mentioned in Chapter 10. In this simpler example, both types of sections have been put into one structure. The structure whose type is **hard** is defined in the usual way as having one char for the shape number, and an unsigned for the period, a total of three bytes. The main structure, **soundit**, consists of a char **sect** which will hold the count number, and an array **section** of type **hard** which will hold the section structures. The main program then calls **setup_sound()** as usual, and assigns values to the sections. The BASIC equivalent of what we want to do is the two section envelope which would be programmed with **ENV 1,=9,1000,1,0,40** and the 'C' version starts with assigning **soundit.sect**, the number of sections as 2. The numbers for the sections are then allocated. For the hardware part the numbers 9+128 and 1000 are assigned as you might expect, but for the software section things are not quite so straightforward. The software section which in BASIC uses numbers 1,0,40 will still use the 1 as its first number, but the other two have to be written in unsigned form. This means multiplying the third number by 256 and adding it to the second. Since $40*256+0=10240$, that's the number we put in as the 'period' number. We are still using three bytes, but in a different way. The hardware deals correctly with this set of three because the first number is less than 128, signifying a software section. The address of the structure then has to be passed to **S_ampl_envelope**, and everything is ready to play. The envelope will not give correct timing, however, until the duration of each note is put to its minimum figure of \ 1. This, in fact, exercises more control on the envelope duration than the software portion, and makes it much easier to use hardware envelopes.

What about using software envelopes, then? The method is no more difficult than we have looked at so far. If you are using an envelope which is purely a software one then the structure is slightly changed, as Figure 9.20 illustrates.

```

#include stdio.h
typedef struct{
    char count,size,time;
}soft;
struct{
    char sect;
    soft section[3];
}soundit;
main()
{
    setup_sound();
    soundit.sect=3;
    soundit.section[0].count=1;
    soundit.section[0].size=8;
    soundit.section[0].time=2;
    soundit.section[1].count=5;
    soundit.section[1].size=255;
    soundit.section[1].time=4;
    soundit.section[2].count=2;
    soundit.section[2].size=251;
    soundit.section[2].time=2;
    S_ampl_envelope(1,&soundit);
    play("Y\1 C\1.D\1.E\1.F\1.G\1.A'\1.B'\1.
C'\1.",2);
}
#include ?basic1.lib?
#include ?stdio.lib?

```

A software envelope, and the structure which is needed to define it.

FIGURE 9.20

The three bytes in each section of a software envelope represent, in order, the step count, step size and pause time. Each of these can be a single byte character, so that structure **soft** has now been defined as containing three characters. This involves one slight complication – the use of negative numbers of step size. The number that has to be used to represent a negative step size is **256-step_size**, so that a step size of -1 is put in as 255, and a step size of -5 is put in as 251. The example then shows a three-section envelope being set up and used. This particular envelope has a curious echo effect that will sound quite different with different lengths of notes – try it out for yourself!

This brings us to the end of this section, with a huge variety of sound effects unexplored. We haven't looked at noise, nor at multichannel music and synchronisation. The reason, apart from lack of space, is that all of these features are programmed in much the same way as in BASIC. If you have a sound program which runs in BASIC, then the library functions in **basic1.lib** will allow you to program the same sounds in 'C'. The HiSoft Manual is particularly helpful to you in this respect, by showing the 'C' library equivalents of the sound statements.

Chapter 10

Assortment.

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

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

```
str(j)=s[j];
```

then you can get the report:

```
ERROR 40  
Can only call functions
```

which will certainly draw your attention to the line that causes the problem, but doesn't necessarily remind you of what the fault is. The problem is that round brackets mean a function to the compiler, so using **str(j)** makes it appear that you are trying to assign a value to a function, which is impossible.

Another fruitful source of error messages lies in assignments. You become so accustomed to the few data types in BASIC, that you assume that almost any variable can be equated to any other. In 'C' simple data types are treated quite loosely, and you can play with ints and chars almost as if they were interchangeable. This shouldn't be allowed to go to your head, though, because there is an important difference. The char type needs only one byte of memory, whereas the integer needs two. The importance of this in reading the EOF character has already been mentioned. The other side of the coin is that 'C' treats strings (arrays of characters) rather more strictly than BASIC does. In particular, you cannot assign a string to a variable name by any simple method such as using **string1=string2** as you might in a more BASIC-like language, or even in Pascal. Instead, strings must be assigned character by character, as **strcpy** or **blt** does, or by passing pointers. This is probably one of the items that causes most annoyance to ex-BASIC programmers! In addition, even the most obvious error messages may not be all that they seem. An error message delivered in the middle of a line may refer to the preceding line, or even to a line which is several steps away. The most baffling message is usually the **Missing** type, because whatever is reported as missing may be staring you in the face, on the screen. As the manual comments, this particular message is a shorthand form for several types of faults and you simply have to look at your program listing rather closely.

To anyone coming to 'C' from BASIC, the use of the semicolon to mark the end of a statement is sometimes baffling because of the exceptions. The most important rules concerning semicolons are that there should be none following **while**, **do**, or following the loop statement such as:

```
for (j=0;j<=100;j++)
```

unless this is being used purely as a time delay. You don't need semicolons following */*remark*/* lines, but if you do put in the semicolons the compiler will not object. The program will object, however, if you have put in a semicolon following a **#define**. As the manual points out, using **#define NULL 0** means that **NULL** is defined as **0** throughout the program. If you use **#define NULL 0;** then **NULL** is defined as **0;** which is not a number, and which will cause some very peculiar things to happen. The problem here is that the error is not found at the time when the **#define** line is being dealt with, but later, when you attempt to use **NULL**. The error message will also be one that does not make it clear what has happened, like 'Missing). The worst errors occur when you have done something instinctively. Top among this type of thing is using **x=y** when you mean **x==y** in a test. The statement **if (x=y)** means that the value of **y** is assigned to **x** and if it is not zero then something has to be done. The statement **if (x==y)...** means that something is to be done only if **x** is identical to **y**. The confusion arises because both statements are legal syntax, but with very different meanings.

Other errors are rather easier to spot because they involve actions which you would not be working with in BASIC. Pointers are a notorious source of problems, even for fairly experienced programmers in 'C'. One very common cause of trouble is declaring a pointer, but forgetting to assign some value to it before using it. A less obvious problem is that a string name is a pointer to the string, but an integer or character name is not. In addition, the name of a structure needs the **&** sign if it is being passed to a function. The other common pointer error is to forget that incrementing or decrementing a pointer will

automatically cause a change of the correct number of bytes, according to the variable type. The **scanf** function is another potent source of errors. It's always wise to start a **scanf** control string with a blank, because this prevents problems when the input is done in a loop. By using a blank, **scanf** will skip over any (RETURN)/(ENTER) character which is in the keyboard buffer from a previous entry. At various stages in this book, we have seen what problems this stored character can be when other input functions are used. One minor point about **scanf** is that there is no %u specifier string for unsigned numbers. Using "%u" in a **scanf** input will cause very odd corruption of the other variables. A more common problem is to forget that each variable in **scanf** must be a pointer, so that string names or & with other names will be required.

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

Hints & tips.

There are a few odd hints and tips which can make your 'C' programming career rather easier. Don't forget, for example, the very useful find-and-replace facilities of the HiSoft C editor. As always, if you have programmed only the Amstrad in BASIC you will not have come across these useful aids to programming. If you have used any good word-processor software, however, you will know how much time can be saved by using search and replace actions. In a program which consists of a large number of **printf** statements, for example, you can save a lot of typing time by abbreviating **printf** to **P**. You can make similar economies with **scanf** and **gets**. If you use a lot of these search and replace actions, though, it's a good idea to keep a note of which abbreviation you have used for which purpose. If you don't there is a risk of using the same abbreviation for two different instructions, or of forgetting that you have used an abbreviation. These editing actions are particularly useful, of course, in versions of 'C' which do not use line numbers, but if you do not have a printer it's very handy to be able to find, for example, which line contains something like /*FIND DATA*/. In addition to the use of the editor, of course, you can make use of **#define**. You can, for example, use a line such as:

```
#define P printf
```

to make each **P** in the program act as a **printf**. This is less desirable than using the editor, however, particularly if you intend to print out your program at any stage. The reason is that it makes the 'C' statements look non-standard, and so much more difficult to read. It's better to keep the use of **#define** for quantities such as **NULL**, **EOF** and the like, which can vary from one system to another but which should be reasonably standardised in programs.

Other actions.

Inevitably, when you make use of a language some parts of that language will come in a lot more use than others. In the course of this book, I have tried to put the greatest emphasis on the features of 'C' that you are likely to spend most time with. This emphasis has meant that other features have been omitted or lightly glossed over, and in this section we'll try to remedy that deficiency. It's possible that none of the actions which are described here may ever interest you. On the other hand, one or two of them might just be exactly what you have always needed but were afraid to ask about. Particularly belonging to this section are the statements and operators that apply to binary numbers and to machine code. You will find that many of the functions in the **basic** libraries are written by using calls to the firmware. If you are a proficient machine-code programmer you may wish to write different functions for some applications, depending on your interests. You might, for example, want to add a **circle(pos,rad)** function, or a **sound** function which is more similar in syntax to the BASIC version. All of this is possible, and it makes 'C' a particularly useful language which is never static. HiSoft have promised to release new libraries as more functions are added, and this is a very desirable method of supporting the language. In addition, the C User Group in the U.S.A. keeps discs full of source-code routines, though these are not at the moment available in the unusual Amstrad 3" disc format. If you can make use of larger discs, however, it's possible that you could read the ASCII text from these discs, which consist of 'public-domain' software. Public domain means that no copyright is involved, and the programs are free though you need to pay for discs, copying time, postage etc. It would help if these programs could be made available as listings, but unfortunately they are not.

We'll start with a keyword that you may have noticed in the list, but which we haven't used. This takes very little time, because **entry** is simply a spare word which has no effect on any compiler at present. It is a word for an action which is not implemented, and might never be. It should not be typed into a HiSoft C program, as the compiler will reject it. A much more important topic concerns coercions and **cast**. As you will have seen, 'C' treats characters and numbers in a fairly interchangeable but well-defined way. This means that the result of an expression which uses mixtures of items will be obtained by using a set of rules. These rules are summarised in Figure 10.1, and Figure 10.2 illustrates them in action.

1. Type **char** is always converted into type **int**.
2. If an **unsigned** number is used in an expression, the result is always unsigned.
3. If the numbers are integers, the result is also integer.
4. Any change to this scheme can be made only by using **cast**.

The rules for coercion of number types.

FIGURE 10.1

```

main()
{
char c;
int j;
unsigned p;
c='a';
j=50;
printf("\n%d",j+c);
p=36;
p+=50+c;
printf("\n%u",p);
}

```

A program which illustrates coercion in action.

FIGURE 10.2

Three variable types are declared, and arithmetic is carried out. The addition of a character to an integer produces an integer result, the sum of the integer value and the ASCII value of the character. The sum of the unsigned number, integer and character produces an unsigned number because the character converts to integer, and the sum of an integer and an unsigned is always unsigned. Assignments, however, will convert to whatever form is called for if possible. For example, if you declare **int j** and **char d** and then assign 65 to **j**, then **d=j** can be performed and will result in the character **d** printing as an A, ASCII 65.

These coercions are completely automatic, but there are times when we want to make coercions that are not automatic. We may, for example want to make a pointer into an unsigned number or an unsigned number into a pointer. This latter action is particularly useful if we want to perform the equivalent of PEEK or POKE, as the library illustrates. For such actions, the **cast** action is provided. Note that the use of the word **cast** is a feature of HiSoft C, and does not appear in all other versions. In many varieties of 'C' the **cast** operation is carried out by using the syntax of **cast** (with brackets), but without the word 'cast' being used. This makes it only too easy to perform a **cast** action unwittingly, and the specific use of **cast** is a useful enhancement to the language. The syntax is of the form:

j=cast(type)variable

in which the brackets enclose the type that you want the number to be cast to.

```

main()
{
char *p;
unsigned j;
*p='d';
printf("\n%c",*p);
j=cast(unsigned)p;
printf("\n%u",j);
}

```

Illustrating the use of **cast**.

FIGURE 10.3

Figure 10.3 illustrates the use of cast in forcing a pointer to be cast to an unsigned number. This is not likely to be needed because you want to perform arithmetic on pointers, since you can perform most of the arithmetic actions that you need to on pointers in any case. It's much more likely that you need to assign a definite value to a pointer.

More complications.

The data types that are supplied by 'C' are fixed, and you are not allowed to make up your own data types as you are in Pascal. You can, however, assign names of your own to types by using **typedef**.

```
main()
{
  int j;
  typedef int *p_int;
  p_int point;
  j=45;
  point=&j;
  printf("\n%d", *point);
}
```

Using **typedef** to declare a pointer type.

FIGURE 10.4

Figure 10.4 shows a typical example. The line:

```
typedef int *p_int;
```

means that we can now use **p_int** as if it were a variable type like **int** or **char**. It is, in fact, a pointer to an integer, and by declaring **p_int point** we make the word **point** a variable of type pointer to integer. We can therefore assign to **point** a pointer value, the address at which the value of **j** is held, and we can print the value held in this address by printing ***point**. This action of **typedef** can be very useful, but if too many types are defined it can make a program difficult to follow.

Typedef does not create new types but there is one type that we haven't considered so far, the **union**. A **union** is a 'hold-anything' variable, one that can be used to hold a character, integer, string or whatever we like. It sounds marvellous, but in fact it's not used as much as you might expect. A **union** has to be declared in very much the same way as a structure is declared, using a pattern of the form shown in Figure 10.5.

```

union boss{
    char c;
    char *s;
    int j,k;
} chief;
main()
{
    chief.c='A';
    printf("\n%c",chief.c);
    chief.s="STRING";
    printf("\n%s",chief.s);
}

```

Creating a **union**, a form of variable that will hold any one of a set of types.

FIGURE 10.5

In this example, the type union is declared with the pattern name of **boss**. The union can contain a character, a pointer, or an integer named **j** or **k**. Note that this is one *or* another. A structure, by contrast, contains all of the types that are specified, the union contains any one. In the lines that follow, a type is assigned and its value then printed out. The important point is that you can assign only one value at a time, and you must select the correct name, such as **chief.c**, **chief.j** or whatever is needed. When a union variable is declared, it will reserve as much space as is needed for the largest of its contents. If you make a union type, for example, which contains a character, an integer and a four-character string, then the string is the longest member and will make the union four bytes long, assuming that the total string length is four. A structure would need one byte for the character, two for the integer, and four for the string, a total of seven bytes. If you attempt to print out data which has not been assigned to a union, you will get garbage. For example, if you have assigned a string to pointer **s** in the union, then trying to print out a character **chief.c** or an integer **chief.j** will produce results which may be useful, but usually are not. In this particular example, the attempt to print **chief.j** will usually produce the pointer address **chief.s**.

Finally in this particular collection of less-used statements, we come to the least-used of all, **goto**. Having to use **goto** in a 'C' program is the programmers equivalent of driving a car with a sign on the back that says 'Take care, wally driving'. The use of GOTO in BASIC has little enough justification, because Locomotive BASIC has the WHILE...WEND loop. In 'C', the use of **goto** is useful mainly when you want to try out a loop by a piece of quick editing. Once you have proved the point, you would normally want to make a more permanent form of loop using **while**, **do** or **for**. Of course, once your program has been compiled into machine code with **#translate** no-one will ever know. The use of **goto** requires a label name, which is used to mark the start of the loop or the place to which **goto** leads to. Unlike BASIC, which uses the line number as a label, 'C' requires a word to be put in and followed with a colon, as the illustration in Figure 10.6 shows.

```

main()
{
char *s;
int j;
s="I am a wally...";
j=5;
here:j--;
/*start of loop*/
printf("\n%s",s);
if (j==0) goto there;
goto here;
there;;
}

```

Using **goto**, with its label word.

FIGURE 10.6

The colon can be folowed by a statement, or simply by a semicolon. The effect of the **goto** will be to make the program move to the point which is labelled. The provision of the **break** and **continue** statements in loops provide for practically all the uses that might otherwise justify the use of **goto**, and for that reason it's seldom used.

The # commands.

We have made intensive use of two # commands, **#include** and **#define**, in programs so far. These commands are often called 'pre-processor' commands, because of the way that 'C' is implemented in other computers. In these machines there is a separate piece of program which deals with these commands, using them to alter the text of a program. For example, if **#define NULL 0** has been used at the start of a program, the pre-processor will replace each **NULL** by a zero throughout the program. Only after all this has been done will the compiler get to grips with its work. Similarly, if you have included other files by using commands such as **#include strcmp**, then the pre-processor will get these files (assuming there are such files on the disc) and place their text into yours. In HiSoft C there is no separate pre-processor, and all of this work is done by the main compiler. HiSoft C also allows a few # commands which do not feature in the pre-processor syntax of most 'C' compilers. One of these is **#error**. When you type **#error** into your listing, you release a considerable chunk of extra memory in the computer. This was the memory that was formerly used for the detailed error messages, as distinct from the **ERROR** number messages. The use of **#error** is therefore a useful method of squeezing in a very long program, but the messages will remain suppressed until you switch off and re-load HiSoft C. Unless you can find where the messages are stored and how their presence is indicated, there isn't much choice about this.

The **#direct** command works in a quite different way. To use **#direct**, you must first have typed **c** and (RETURN)/(ENTER). This puts the compiler in action, and typing **#direct+** then puts it into direct mode. This means that any statement which you type will be executed when you press (RETURN)/(ENTER). You must be careful not to press (RETURN)/(ENTER) until you have finished typing statements, which means that you simply type statements separated by semicolons, not in lines. This is most useful for items that are self-contained, like **rawout(4);rawout(1);**, rather than for extended statements. It's handy if you want to send codes to a printer, or to make use of some peeks and pokes or rawout statements. You can get back to normal compiler use by typing **#direct-**, using (ESC) – or by making a mistake in a statement that is to be executed directly.

The other common **#** command is **#list**, which once again is a feature of HiSoft C. Using **#list-** suppresses listing while a program is compiling, and using **#list+** restores the listing. You can see this action at work when you make use of the routines from the libraries. It speeds up compiling to some extent because the screen print routines of the Amstrad machines are rather slow, and in any case once a program is debugged you don't particularly want to see it listing each time it compiles. It's a good idea, then, on your longer programs to incorporate a title using **/*remark*/** lines, and suppress listing immediately after this until the end of the program. That way the heading shows what program is compiling, but the actual listing does not appear.

Statics in functions.

The HiSoft C manual encourages you to make use of static variables wherever possible, because of the saving in memory. This is not the advice that you get with other versions of 'C' but since the Amstrad is, by comparison with the machines on which 'C' was developed, of modest memory size, the advice is sound. Because static is advised as the normal memory type, it's easy to forget precisely why it was evolved and how it can be used. Figure 10.7 is a reminder.

```
main()
{
  int j;
  for (j=1; j<=5; j++)
    stadem();
  stadem()
  {
    static int k=1;
    printf("\n%s %d", "k is ", k);
    k++;
  }
}
```

A simple program to illustrate the action of a static variable in a function which is called several times.

FIGURE 10.7

In this example, a function is called five times. In the function, the static variable **k** is initialised to 1, and its value is printed, then incremented. When the program runs, however, you will see the incremented values of **k** appear in a count-up. The effect of declaring **k** as static and initialising it to 1, is to make it have this value of 1 the first time the function is called. At the end of the function, however, the value of **k** is preserved – this is what a static variable is all about. When the function is called again the initialisation step is ignored, and the value of **k** which is stored from the previous time is used. The value of **k** cannot be used in the main program, because it does not exist in the main program. Even if you have a variable called **k** in the main program, it will have no effect on the **k** that is used in the function. This aspect of a static variable is very important if you want to make use of a function in which a variable is changed. You can avoid the change if you carry out a separate assignment. If, for example, you follow the declaration of **k** with the line:

```
k=1;
```

then the printout from the program will show that **k** has always been 1. For many purposes, however, the automatic storage of variables can be very convenient. In general, programs written in 'C' for other computers are likely to use static variables only where this type of action is wanted, and you may have to be careful if you write these programs again for HiSoft C, using mainly static variables.

Another action which is sometimes needed is a **variadic function**. The functions that we normally use have a known and fixed number of arguments, like **action(a,b,c,d)**. We sometimes need, however, to pass a variable number of arguments. Many textbooks show this applied to **main()** itself, but this is not something that you can do with HiSoft C. The header **stdio.h** shows this action used for two functions, **max** and **min**, which are designed to find the maximum or minimum value, respectively, in a list of numbers. The argument of each function, then, will consist of as many items as there are numbers. The essential feature of a function of this type is to follow its header name with the word **auto**. This indicates to the compiler that the function will be of this 'variadic' type, and that the number of arguments is variable. The output of the auto actions will be to supply two special arguments for the function which are called, by convention, **argc** and **argv**. The function itself is written with one argument in the brackets. This argument, unlike any we have met before, doesn't have to be declared anywhere because it is a special quantity, the number of bytes of argument which have been used when the function was called. Suppose, for example, that you were dealing with integers. Since two bytes are used for each integer, ten numbers in an argument would require 20 bytes. By convention an extra pair of bytes is always held available, making 22. This is because the use of **argc** and **argv** in HiSoft C is not quite the same as its use in other versions. In many versions of 'C', **argc** and **argv** are used to pass commands, with arguments, directly to **main()** so that you can call upon a compiled main program by using its name and passing some values. This is not possible when the program has to be compiled each time it is used. In the original method, **argc** is the number of arguments, and **argv** is a pointer to an array of strings, one string per argument. In this method, **argv[0]** is the program name itself, forming one of the strings, so that a program with no arguments still has an **argv[0]**, and its **argc** value is 1. The first real argument is **argv[1]**, making **argc** equal to 2 and so on. On this basis, **argc** always contains one item more than the total number of items. If the arguments are integers, then this extra **argc** value corresponds to two bytes.

```

total(number) auto
{
static int argc,*argv,t;
argc=number/2-1;
argv=&number+argc;
while(argc--){
    t+=*argv;
    --argv;}
return t;
}
main()
{
int j;
j=total(1,6,4,8,6,2);
printf("\n%d",j);
}

```

A **variadic** function, in which the number of arguments is variable.

FIGURE 10.8

In the program of Figure 10.8, then, we start by defining a *variadic* function called **total**. The program must place this function before any call that is made to it, even though it is, in this case, a function that returns an integer. The header of the function contains only the argument **number** which will be the number of bytes in the argument when the function is called. This quantity **number** does not have to be declared. Instead, we declare the integers **argc** and **t**, and the integer pointer **argv**. Because **number** consists of one item more than the actual list of arguments, we find the item-count from $\text{number}/2-1$. The pointer **argv** then has to be set. The arguments are stored in memory addresses which decrement, so that the address that we get from **&number** when we read this quantity is the lowest address at which the last argument is stored. To get to the first argument, at the top end of the memory range, we need to add **argc**, the item count. The items can then be dealt with in a loop that decrements **argc** and **argv** together. In this example, the argument numbers are added to the totalling integer **t**, so that the total can be returned. If you want to see what is going on, add some lines that will print values of **number**, **argc**, **argv** and ***argv** at intervals.

Binary and machine code.

For many purposes, binary numbers and machine code are unnecessary and undesirable. After all, the whole point in inventing higher level languages like 'C' was to avoid the chore of programming in machine code or assembly language, and having to think in terms of binary numbers. Nevertheless, there are times when we have to do just that, either in order to save memory, devise some very clever programming or speed up the execution of something. The standard 'C' language provides several operators for use with binary numbers, and the HiSoft version also provides a new reserved word, **inline**, which has the effect of entering machine-code routines directly into memory. The provision of **inline**, incidentally, makes it comparatively easy to write a Z-80 assembler for yourself. That's just the type of task that 'C' was intended to make easy.

We'll start with the operators that carry out binary logic comparisons of two numbers. The three binary operators of this type are & (AND), | (OR) and ^ (XOR). Note that these are single characters, and you must be very careful to distinguish the & from &&, and the | from ||, because these are used in very different ways. If you aren't familiar with binary logic operators this is no place to start learning, and I suggest that you skip the rest of this chapter until you have had time to digest a book on binary numbers and machine code.

```
main()
{
char c,d;
c=0xA6;
d=0x3C;
printf("\n%s %x","AND gives ",c&d);
printf("\n%s %x","OR gives ",c|d);
printf("\n%s %x","XOR gives ",c^d);
}
```

Using some of the binary operators. These are the three operators which make a logical comparison of binary bits.

FIGURE 10.9

Figure 10.9 illustrates the binary operators being used to compare two numbers. The numbers are written in hex code, using the **0x** prefix which is used to mark hex numbers in HiSoft C. A number written normally is always taken to be a denary (ordinary scale of ten) number unless it is being used along with a backslash. Numbers which are preceded with **0**, or as characters with a backslash, are taken to be in octal code. Because of this use of the zero, you must be careful not to start any ordinary denary number with a zero. In this example, the numbers are hex and are prefaced with the **0x**. The numbers are declared as characters which means that we are working with eight bits only, and the three **printf** lines give the results of AND, OR and XOR operations carried out bit by bit. The **printf** lines make use of the **%x** specifier to print out the results in hex. In case you are dubious about these results, Figure 10.10 shows an analysis of what is happening.

Hex A6= 10100110 in binary
Hex 3C= 00111100 in binary
AND gives 00100100 which is hex 24.

Hex A6= 10100110
Hex 3C= 00111100
OR gives 10111110 which is hex BE.

Hex A6= 10100110
Hex 3C= 00111100
XOR gives 10011010 which is hex 9A.

An analysis of the binary number actions.

FIGURE 10.10

HiSoft C does not permit some of the 'bit field' actions of the 'C' standard, but the equivalent effects can all be obtained by suitable use of these operators. For example, if you wanted to check that bit 4 in a byte was set, you could use a test like:

```
if (c&&0x10==0x10).....
```

and you can make use of the usual 'masking' effects of numbers such as **0xF0** and **0x0F**. These bitwise operations can be used, of course, on integer numbers as well. The integer is stored as two bytes, so that the results of these operations will be expressed as two byte numbers unless they can be fitted into a smaller space. Remember that small negative denary numbers (like -5) will be stored as a two-byte integer, with the upper byte equal to **0xFF**, so that bitwise comparisons with such numbers will often result in four-figure hex numbers.

In addition to these operators which compare two numbers, there are two shift operators which operate on a single number, integer or character. The left shift is indicated by `<<`, which must be followed by a number that gives the number of places shifted. Using `<<1` will cause a left shift of one place, using `<<3` will cause a left shift of three places. These left shifts are logical shifts, with zeros being used to fill in the right hand side. The shift will change the most significant bit of the upper byte if necessary, so that the apparent sign of an integer can be changed by a shift of an integer. The right shift is signalled by using `>>`, again followed by a number. This is an 'arithmetic' type of shift because the most significant bit of the upper byte is not shifted, and the shift action copies this byte at the left hand side rather than putting in zeros. In other words, if the integer starts with the bits 10.., then a right shift of one place will make this 11.. rather than the 01 which would result from a logic shift. The right-shift action therefore does not change the sign of a number.

Some shift actions are illustrated in the program of Figure 10.11.

```
main()
{
char c,d;
c=0xA6;
d=0x3C;
printf("\n%s %x","shift left c- ",c << 1
);
printf("\n%s %x","shift right c- ",c >>
1);
printf("\n%s %x","shift left d three pla
ces- ",d<<3);
printf("\n%s %x","shift right d three pl
aces- ",d>>3);
}
```

The use of shift actions on binary numbers.

FIGURE 10.11

The two numbers are declared as characters and one is shifted by one place left, then by one place right. The other number is shifted in each direction by three places. In each case the results are printed in hex, and you can see the effects of type coercion working to display two of the results as integers rather than characters. Figure 10.12 shows an analysis of these shifts.

Hex A6 is 10100110 in binary

When this is shifted one place left, with zero added on the right hand side, it becomes:
101001100 which is hex 14C.

Hex A6 is 10100110 in binary

When this is shifted one place right, it becomes:

01010011 which is hex 53.

Note that the left hand bit has changed – the rule about the left hand bit not changing applies only to 16 bit numbers.

Hex 3C is 00111100 in binary

When this is left shifted by three places, it becomes:

00011110000 which is hex 1E0.

Hex 3C is 00111100 in binary

When this is right shifted by three places, it becomes:

00000111 which is hex 07.

Sixteen-bit shifts.

Hex C03C is 1100000000111100 in binary.

When this is left shifted by three places, it gives:

0000000111100000 which is hex 01E0.

Hex C03C is 1100000000111100 in binary.

When this is right shifted by three places, the most significant digit of 1 is retained, and copied into the shifted places. This gives:

111110000000111 which is hex F807.

Analysing the effect of shifts.

FIGURE 10.12

If you now modify the program to declare **int d** and assign the number **0xC03C** to **d**, you can run the program again to see the effect of left and right shifts on a number which has its most significant bit set. The left shift behaves as a normal logic shift, giving **0x01E0**, and changing the sign of the number if it is printed as a denary number. The right shift copies the most significant bit of '1' into the the next three places, and gives the result **0xF807** of the same sign as the original number. These are also shown analysed in Figure 10.12. You will find the shifts widely used in the library routines as a method of multiplying or dividing by 2, or for bitwise analysis.

Inserting machine-code.

The **inline** reserved word of HiSoft C is not a standard 'C' word. It is, however, a very useful enhancement of standard 'C', because it allows you to write library functions which include, or are made exclusively of, machine code. Since all of the facilities of the computer are available from machine code, this allows you to write, for example, a **circle(centre, radius)** function or new methods of controlling the sound system as you please. Obviously, it helps considerably if you know how to write and use machine code, but even if you are not an expert with machine code it can be useful. You can, for example, make use of small pieces of machine code which you see printed in magazines, and transform them into 'C' library functions. You need to know what you are doing, and the code must be the type that is 'relocatable' meaning that it can be written in any part of the memory. The **inline** statement does not allow you to choose memory addresses for your machine-code, it simply places it 'in line' with the rest of the code that the 'C' compiler generates, hence the name.

```

main()
{
    fill();
}
fill()
{
    inline(
0xCD,0x6C,0xBB
0x01,0xE7,0x03
0x3E,0x41
0xCD,0x5A,0xBB
0x0B,0x7B,0xB1
0x20,0xF6
);
}

```

Using **inline** to insert machine code into a function. No RET code must be used.

FIGURE 10.13

Figure 10.13 shows a very simple example of **inline** in use. The machine code program is a trivial one – it simply fills the screen with the letter ‘A’. The important point is how it is introduced into the ‘C’ program, and how it is terminated. Normally, a machine code program ends with the RET command, **0xC9**, or with some variation on this command. This should not be done when **inline** is used, because the RET will cause a lockup unless it is part of a subroutine within the code. You have to be careful about some looping programs in machine code which might include conditional returns such as RET Z. The best way of adapting these is to change the conditional return into a jump to the last byte of the program, which can then be a NOP byte of 00. No such complications arise in this example, however, and the code is written following the **inline** word, with the code enclosed in brackets. Note that the code can be written as shown in separate lines, with commas separating the code bytes. No semicolon must be used until the final bracket has closed on the codes. The compiler will then assemble the code as it comes along, and place it as part of the compiled ‘C’ code. In this example, when you run the program, the screen fills with the letter ‘A’, and the final line then scrolls to allow the ‘Type y to run:’ message to appear.

```

unsigned _h1;
main()
{
    int n,j;
    for (n=0;n<=100;n+=2)
    {
        side(n);
        for (j=1;j<=1000;j++);
    }
}
side(n)

```



```

{
  _hl=n;
  inline (
    0x2A, &_hl,
    0xCD, 0x05, 0xBC
  );
}

```

Passing a parameter to a machine code routine which is written **inline**. The routine performs a sideways scroll, using the position number which is passed from the main program.

FIGURE 10.14

Figure 10.14 shows an advance on this technique. This time a parameter is passed to the machine-code routine, using the method which is normal for HiSoft C. Unlike the BASIC method this depends on the use of a variable as a temporary store, and this variable, `__hl`, is declared as unsigned before the start of the main program. The machine code itself is a sideways scroll routine, which requires a position number into the HL register pair, then a call to `0xBC05`. The important point here, then, is the method of getting the parameter value `n` into the HL register pair. Normally when you interface from BASIC to machine code, the first part of your machine code program consists of reading the parameter bytes using the IX registers. The HiSoft manual points out that the IX register is also used when a function is called, but the passing of the parameter must be done in a different way. The method is to pass the parameter to the unsigned number `__hl`. Obviously, you could call this whatever you like and the `basic1.lib` routines use `reg__hl`, but whatever you use it's a good idea to have a name that reminds you of what it's to be used for. The number stored in this variable then has to be passed to the routine and, in this case, it's needed in the HL register pair before the shift routine is called. The transfer is done by using the code for LD HL, which is `0x2A`, and following this inline with `&__hl` to get the address of the variable. The call to `0xBC05` is then made in the next line of codes and, as usual, the set of **inline** codes ends with no RET byte. If, incidentally, you require an **inline** function to return an integer or pointer, this can be done by loading the required bytes into the HL and BC registers before the routine ends.

This concludes our tour through the facilities of HiSoft C. Like any other language 'C' takes some time to learn, and a lot of practice is needed to become really familiar with it. It also requires you, like any other language, to write programs of your own design to become really familiar with the feel of the language. In my programming time I have had to use Fortran, Basic, Pascal and C, and of these 'C' is the one that is always the greatest joy to return to and write in. It's difficult to say why, but I think it's best summed up in the word 'fascination'. 'C' is a fascinating language to use, not least because almost every action that can be carried out with the hardware you have can be programmed with 'C'. The other side of the coin is the lack of safeguards – that a 'C' program can suffer from an obscure bug which may keep you up all night trying to swot. To me, that's part of the fascination, the pitting of wits against the inexorable logic of the machine and the language. I shall never tire of it, and I hope that I have been able to pass some of my own sense of fascination to you.

Appendix A

Binary, Octal and Hex codes.

Throughout the history of computing, programmers have used various forms of number codes in preference to the normal scale of ten. The reason is basically that computers store numbers in memory units, each of which is a type of switch. A switch can be either on or off, and so each unit of a memory can store only two codes, one for on and one for off. We can put this into number terms by using off to mean 0 and on to mean 1. With only these digits to use, then, all numbers have to be stored and manipulated inside the machine using binary code, which has only the two digits 0 and 1. Using only two digits makes no great difference to the way that we write numbers, however. In denary (scale of ten) we count up from 0 to 9, and then the next number is written by placing a '1' in another column, the 'tens' column, and a zero in the units column. In binary the count in the units column is from 0 to 1, and then the next number is 10, – one 'two' and no units. The number three is then represented as 11, a two and a unit, and four needs another column, 100. Figure A.1 illustrates the sequence of these numbers, and shows how to convert between binary and denary numbers.

Position values –

Bit.No.	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

For each further place to the left, use a position number which is double the previous one.

Binary count from 0 to 15 denary.

Denary	Binary	Denary	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Conversions:

1. Denary to binary.

Divide the number by 2, and put the remainder next to it. Do the same with the result of the division, and so on until the last number is zero. Then read the remainders *from the bottom up*.

Example: Conversion of 58 denary.

Number – 58 divide by 2=29 and 0 over.
29 divide by 2=14 and 1 over.
14 divide by 2=7 and 0 over.
7 divide by 2=3 and 1 over.
3 divide by 2=1 and 1 over.
1 divide by 2=0 and 1 over.

Reading from the bottom of remainders gives 111010, which is the binary equivalent. This can be padded out with zeros on the left hand side to make it an 8-bit or a sixteen-bit number. For example, as an 8-bit number, it would be 00111010.

2. Binary to denary.

Write the binary number with the position values above the '1' digits, then add the position values. For example, the binary number 01011011 is written as:

	64		16	8		2	1
↓	↓	↓	↓	↓	↓	↓	↓
0	1	0	1	1	0	1	1

The position values added give $64+16+8+2+1=91$.

Binary numbers, and conversion between binary and denary.

FIGURE A.1

The trouble with binary numbers is that they contain a lot of 1's and 0's, quite a dazzling number if your computer happens to use them in groups of 32 or more. Small computers use only eight binary digits (or **bits**) at a time, but even so, the sets of 1's and 0's can hypnotise you into making silly mistakes. Because of this programmers have devised other code systems, of which the most common are octal and hex. For microcomputers octal is very seldom used, and its use in HiSoft C is the first I have ever encountered on a micro.

The trouble with denary, you see, is that ten is not a power of two, as 4, 8, and 16 are. This makes it much more difficult to convert easily between denary and binary. Octal is based on a scale of eight, and its main feature is that conversion between octal and binary is very simple, as Figure A.2 shows.

Octal count 0-7 as in denary, then-

Denary	Octal
8	10
9	11
10	12
:	:
16	20

etc.

Conversion of denary to octal:

As for denary to binary, but dividing by eight. For example, denary 467 is converted as follows:

- 467 divide by 8=58 and 3 over.
- 58 divide by 8=7 and 2 over.
- 7 divide by 8=0 and 7 over.

This makes the octal number equal to 723.

Octal to denary.

As for binary, but use the place numbers:

32768 4096 512 64 8 1

For example, the octal number 421 is $4*64 + 2*8 + 1 = 273$ denary.

Octal and Binary.

Each octal number corresponds to three bits of binary, from 000 to 111. See the binary number table for these equivalents. To convert octal to binary, simply write down the three-bit binary equivalent of each octal digit. For example, octal 254 becomes:

2 5 4
010 101 100

giving the binary number 010101100.

For converting from binary to octal, divide the binary number into three-bit groups, starting from the right hand side. Convert each group into octal, including any one or two bit number at the left hand side.

For example, the binary number:

1001110110100110 is grouped as:

1 001 110 110 100 110

giving

1 1 6 6 4 6 octal.

Octal numbers, and their relationship to binary.

FIGURE A.2

The digits of an octal scale are simply the digits 0 to 7 with the new column being used for each power of 8, such as 8, 64, 512 etc.

Hex, or hexadecimal, is even better from the programming point of view. One single hex digit will represent a number which uses up to four binary digits. This makes the system particularly suitable for modern microcomputers, which use groups of 8, 16, and 32 bits almost exclusively. Since a scale of sixteen is used, we need digits for 0 up to denary 15, and for the numbers denary ten to denary fifteen we use the letters A to F, as Figure A.3 shows.

Denary	Hex
0	00
1	01
etc.	to..
9	09
10	0A
11	0B
12	0C
13	0D
14	0E
15	0F
16	10
17	11

etc.

The hexadecimal (hex) code.

FIGURE A.3

The conversions between hex and binary are particularly simple, as Figure A.4 illustrates.

Hex	Binary	Hex	Binary
00	0000	08	1000
01	0001	09	1001
02	0010	0A	1010
03	0011	0B	1011
04	0100	0C	1100
05	0101	0D	1101
06	0110	0E	1110
07	0111	0F	1111

To convert hex to binary, write the equivalent four-bit binary number for each hex digit.

For example, to convert Hex AF to binary, write down the codes 1010 for A and 1111 for F giving 10101111 as the binary equivalent.

To convert from binary to hex, group the binary digits into fours starting from the right hand side. Then write the corresponding hex digits, not forgetting any one, two or three digit group at the left hand side.

For example: Binary 11001011011 is grouped as 110 0101 1011, so that in hex it is 6 5 B.

Converting between hex and binary.

FIGURE A.4

Finally in this brief summary, we come to the problem of negative numbers. There is no provision for the use of a negative sign in binary, octal or hex. The system that is used is to make the most significant bit (left hand bit) of a binary number act as a sign bit. If we use 16-bit numbers, for example, like the int type of 'C', then a number 0111111111111111 is positive, and the number 1000000000000000 is negative. Figure A.5 shows how the binary equivalent of negative numbers can be found, and how the conversion in the other direction is achieved.

For integer numbers of two bytes, the left hand bit (bit 15) is used as a sign bit. This means that the number 0x7FFF is the largest positive integer, denary 32767. The number 0x8000 is equivalent to -32768, the largest negative number. In denary terms, to find the equivalent of a negative number subtract the number value from 65536, and then convert to octal, hex or binary. When converting back, a negative result should be converted in the same way.

For example, converting -76 gives $65536 - 76 = 65460$, which in hex is FFB4, and in octal is 177664. In binary, this is 111111110110100.

Converting the binary number 1100111101101011 into hex gives CF6B, octal 147553, and denary 53099, so that the number it represents is -12437.

Negative numbers in binary code.

FIGURE A.5

INDEX

A

arrays 4-4
arithmetic heirarchy 2-9
argument 3-2
assignment 1-2
atoi() 2-16, 3-5
automatic variables 2-14

B

backslash 2-8
backup 2-1
bad declaration 2-6
bar (|) 1-7, 5-4
basic1.lib 9-8
basic2.lib 9-2
beep 3-2
binary 10-11, apx A
break 3-12, 5-1
byte 4-1

C

c command 2-6
case 5-1
cast 7-7, 10-4
cent 8-8
centring text 8-8
changing strings globally 1-5
char variables 2-16
clear screen 2-7
coercion 10-4
comments 2-6
compiling programs 2-6, 2-10

compiler 1-1
compound statement 3-11
concatenation 8-14
continue 3-12, 5-6
constants 2-11
curly brackets 2-6

D

d command 1-5
data, structured type 4-4
decimal numbers 2-9
declaration of variables 1-1, 2-14, 7-16
default 5-2
#define 2-12
deleting lines 1-5
denary numbers 2-9
disc commands 1-7, 5-8
division 3-7
do... while 4-10
draw 9-3

E

editing 1-4, 2-7
else 3-9
end of statement (;) 2-2
enter key, ignoring 5-4
entry 10-4
envelope, sound 9-16
exchange 7-4
expecting a primary here 2-7
extern variables 2-15, 3-15

F

f command 1-5
false 3-9
fault-finding 10-1 et seq.
fclose 5-9
fgets() 5-14
field 4-6
fielding 2-13
filing commands 1-5, 1-7
filio() 6-7
finding strings 1-5, 8-16, 8-18
floating point 2-11
fopen 5-9
formatting 2-6 et seq.
form feed 2-8
fprintf 5-9
fscanf 5-9
function 1-3, 3-1 et seq.

G

g command 2-1
G_clear_window 9-2
G_line_absolute 9-2
G_move_absolute 9-2
G_set_pen 9-2
getc() 5-10
getchar() 2-15
getting files or data 2-1, 5-9
global find/replace 1-5
goto 10-7
graphics 9-1 et seq.

H

hexadecimal apx A

I

i command 1-4, 2-7
identifiers 2-4
if test 3-9
#include 1-6, 2-6

(x)

selective 3-3
indexed variables 4-4
initialisation of variables 2-14
ink 9-8
input
disc/tape 2-1, 5-9
keyboard 2-15
tape/disc 2-1, 5-9
instr 8-16, 8-18
int 1-2
integer 1-2
division 3-7
interpreted language 1-1
isalpha 5-6
isspace 5-6

J

justification 2-9

K

keywords 2-2 et seq.
keyboard input 2-15
keyhit() 5-11

L

l command 2-6
left\$ 8-9
left justification 2-9
library functions 1-1, 3-3
line numbers 1-3 et seq.
linked lists 7-14
#list 3-5
listing programs 1-6, 2-6
lists, linked 7-14
loading 2-1, 5-9
local variables 2-14
logical or 5-4
loop 1-1

M

machine code 1-6, 10-11
main function 2-2
mathematical heirarchy 2-9
maths symbols 3-8
max 3-5
mid\$ 8-9
min 3-5
missing error 10-2
mode (screen) 2-17, 3-3

N

n command 1-5
names 2-4
new line 2-9
not operator (!) 3-8, 5-8
number specifiers 2-9

O

octal code 2-8, 9-3, apx A
operators 3-8
or (| |) 5-4
output
disc/tape 1-5, 1-6, 5-8
printer 1-6
screen 1-6
tape/disc 1-5, 1-6, 5-8

P

p command 1-5
Pardon? error message 1-5
pass 1-1
percent sign 2-9, 3-7
play 9-10
pointers 4-1 et seq., 5-9
poke 3-3
portable code 1-6, 2-4
predefined identifiers 2-4
pre-definition 1-3
preprocessor 2-12

printat 8-5
printf identifier 2-6
printouts 1-6
putc() 5-10
putchar() 4-10
putting files or data on tape or disc 1-5,
1-6, 5-8

Q

qsort 7-11
queue ,sound 9-19

R

rand 9-2
random numbers 9-2
rawin() 5-7
rawout() 2-16
reading files or data 5-9
records 6-1
recording data 5-8
register variables 2-15
remarks 2-6
renumbering lines 1-5
reserved words 2-5
restriction 2-15
return 3-13
right\$ 8-9
right justification 2-9
running a program 2-6

S

S__ampl__envelope 9-18
saving to disc or tape 1-5, 1-6, 5-8
scanf 4-3
screen
modes 2-17, 3-3
output 2-6
selective inclusion 3-3
semicolon (;) 2-2
separators 2-2
setup_sound 9-14
Shell-Metzner sort 6-19

sorting 6-16
sound 3-2, 9-10 et seq.
source code 1-5
srand 9-2
statement 2-2
static 1-2, 2-14
stdio 3-3
strcat 8-14
strcmp 5-7
strcpy 4-9
string 1-3, 8-1 et seq.
string\$ 8-12
strlen 8-4
strncpy 6-15, 8-4
strpbrk 8-18
strspn 8-19
structure 1-3
structured data types 4-4
structured variables 4-4
subscripted variables 4-4
symbols 3-8
switch 5-1

T

tab 8-2, 8-3
tape commands 1-7, 5-8
test() 3-9
tolower 5-6
Too many operators 6-19
#translate 1-6, 2-10
true 3-9
type command 2-1
typedef 7-16, 10-6
types of variables
 automatic 2-14
 char 2-16
 extern 2-15, 3-15
 indexed 4-4
 integer 1-2, 2-15
 local 2-14
 nested structures 6-26
 pointer 4-1 et seq., 7-1 et seq.
 records 6-1 et seq.
 register 2-15
 static 1-2, 2-14
 string 1-3, 8-1 et seq.

structure 6-1 et seq., 6-26
structured 4-4
subscripted 4-4

U

undefined symbol 1-6, 2-2
undefined variable 2-14

V

variables 2-4, 2-14
 automatic 2-14
 char 2-16
 extern 2-15, 3-15
 indexed 4-4
 integer 1-2, 2-15
 local 2-14
 nested structures 6-26
 pointer 4-1 et seq., 7-1 et seq.
 records 6-1 et seq.
 register 2-15
 static 1-2, 2-14
 string 1-3, 8-1 et seq.
 structure 6-1 et seq., 6-26
 structured 4-4
 subscripted 4-4
vertical space 8-8
vspc 8-8

W

w command 1-6
while 4-10
write 7-3

Programming in 'C' on the Amstrad

The language 'C' has been used for a considerable time by professional programmers using minicomputers, but not until the arrival of Hisoft's 'C' has it been available on Amstrad microcomputers.

This book assumes that the reader is familiar with Locomotive BASIC and frequently compares this with 'C'. However, 'C' has much more to offer than BASIC and many things which are quite difficult to do in BASIC can be quite simple in 'C'. The author has taken every opportunity to show the reader the new ways of programming available and has included many reminders about old BASIC habits that must be abandoned.

Because of the close inter-relationship between the way programs are designed and the way they are written, the author links these topics together. The book has been written entirely around the conventional 'top-down' method of structured programming, a method designed to make complex programs relatively easy to write and understand.

This book is suitable for users of the Amstrad CPC464, CPC664, and the CPC6128.

GLENTOP
PUBLISHERS ■ LIMITED

ISBN 0-907792-86-3

Glentop Publishers Ltd.,
Standfast House, Bath Place,
High Street, Barnet, Herts. EN5 1ED
Tel: 01-441 4130

£8.95



9 780907 792864

GLENTOP

PHILIPPINE LIMITED

CO'S On the Amsterdam Street
Singapore

AMSTRAD

CPC



MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA



<https://acpc.me/>

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.

[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.

[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.