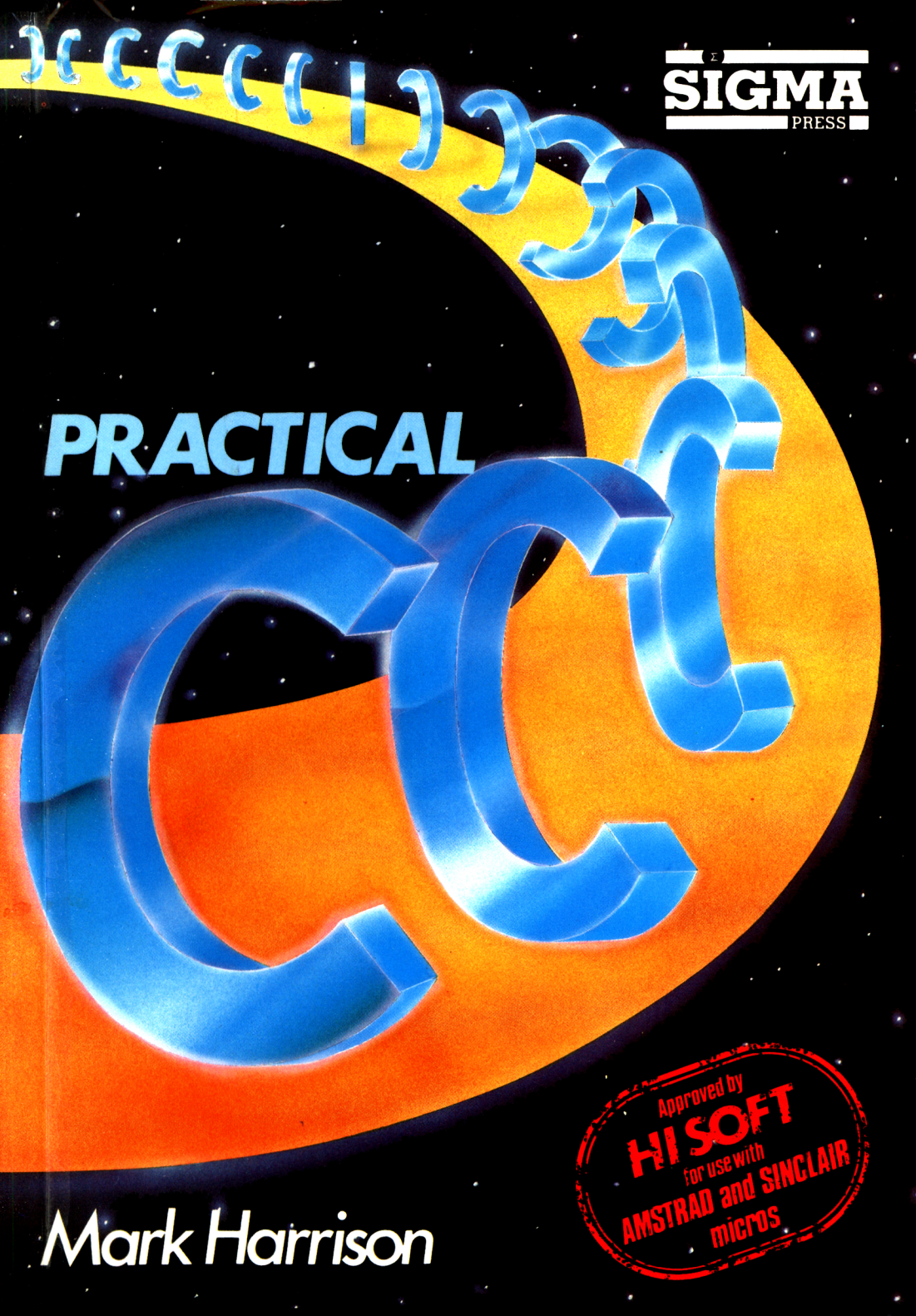

SIGMA
PRESS
PRACTICAL
Approved by
HISOFT
for use with
AMSTRAD and SINCLAIR
micros
Mark Harrison

Practical C

Mark Harrison

SIGMA PRESS

ISBN 1 85058 035 9

Published by:

SIGMA PRESS
98a Water Lane
Wilmslow
Cheshire
U.K.

Printed in Malta by Interprint Limited

Distributors:
U.K., Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester
West Sussex, England

Australia:
JOHN WILEY & SONS INC.
GPO Box 859, Brisbane
Queensland 40001
Australia

Acknowledgements: UNIX is a trademark of Bell Laboratories. CPC 464, 664 and 6128 are trademarks of Amstrad Consumer Electronics plc.

CONTENTS

CHAPTER 1
C—an overview

1.1 Introduction

"I can write BASIC programs but they are far too slow" "I learnt machine code to write efficient programs – but it is so tedious to use" "Machine code!! – far to complicated for me"

These are three views commonly expressed by enthusiasts who have realised the serious limitations of the standard BASIC interpreter supplied with their personal computer. Recently a new computer language, C, has developed a cult following and has become very popular in a short space of time. To have gathered such momentum, so quickly, there must be a very good reason. As you read on you will see the great advantages and power of C and will, no doubt, join the ever growing bandwagon of C fanatics.

The C language was developed at Bell Laboratories in the early 1970s and is an extension of the Basic Combined Programming Language (BCPL). It is often closely associated with the Unix operating system on which it was developed, and to many Unix users the one implies the other. Now Unix is almost entirely written in C reflecting the powerful capabilities of the language.

C is a general purpose programming language that places emphasis on concise programs and flexible expressions. It is a *compiled* language, which means that it is first written in *source* format and then translated into *object* or *machine code*. The object code is then *linked* with other code modules to form the final code version which can be executed by the computer. Because the machine code program is understood by the computer it may be executed directly and so processed fast and efficiently.

Due to the nature of C, it is very easy to write programs in C that, with only minor amendments, will work on any machine supporting C. Thus C, is favoured by many serious software houses, since the time to port their systems onto different machines is considerably reduced (especially if they have the same target operating system). Whilst no language can be considered truly portable, C does approach this ideal.

It is common practice for BASIC programmers to start with an initial idea and then continue to add extra refinements until the logic behind the program resembles "spaghetti" with numerous jump statements sending control in all directions. C enables the better approach of using *structured programming* techniques. This is a system in which programs are broken down into blocks or *functions*, each of which has a single specific purpose. Structured programming also tries to avoid the GOTO statement in an attempt to keep the logic in a program simple – this is easily achieved using other, more powerful, C control statements. Modularising a program in this fashion means that each function can be tested in isolation so that any errors or *bugs* can easily be detected. In addition, amendments and enhancements can be made with minimal changes to existing code, thus helping to avoid the introduction of new bugs.

Function libraries are supplied with C compilers and additional libraries can be created to contain commonly used functions such as database routines, screen handlers, etc. Functions that are specific to any hardware, such as I/O routines, should be isolated in libraries so that porting software onto other machines requires admendments which can easily located.

C is fairly small in size compared with other compiled high level languages, making it suitable for use on microcomputers and easy for the beginner to learn Despite its size, it supports elegant and powerful programs. Developing C programs is not as straight forward as working in BASIC because the user is not protected from crashing the machine. A bug in a BASIC program usually results in an error code, the wrong answer or a loop. C is much more like machine code, where bugs can have wierd and obscure effects causing numerous problems – the most obvious being when the computer gets into a state such that the only escape is to reset it by disconnecting the power supply, thus losing the contents of the memory!!

Since the first implementation of C numerous versions have appeared; notably, for small microcomputers:

> Mark Williams, De Smet, Manx Aztec, Digital Research, BDS, ECO, Lattice, Computer Innovations, Consulair, Codeworks Q/C, Softworks, Microsoft,

to name just a few.

Whereas these implementations require a disk system so that data can be stored permanently on a magnetic meduim, a version of C has been developed by *Hisoft* with small personal computers in mind, enabling the language to be used on systems with a minimum configuration of just RAM and cassette tape. It is however worth mentioning that the use of a disk drive makes life a lot easier.

The Hisoft implementation of C is designed to be as close to the true definition of the language as possible. However, there are a few exceptions and extensions that have occurred because of the specific hardware environment for which it has been designed. Some of these limitations are expected to be removed in future releases.

The Hisoft C compiler is based entirely in RAM, together with an editor, the C program source and the resultant machine code. At the time of writing, versions are available for the Amstrad CPC 464/664 and ZX Spectrum, and are being developed for use on MSX, CP/M and other Z80 based systems. The Hisoft compiler is being produced for large volume sales and so costs a fraction of other C compilers, thus enabling the language to be used by many. Because of this expected popularity, the examples in this book are written in Hisoft C. Any differences between Hisoft C and the true definition will be highlighted for the benefit of readers using other versions of the language.

To get started with your compiler, study the documentation provided to see the specifications and differences from the true definition of C. Find out how to create and compile source files, use the editor and library facilities and link different code modules together to produce an executable program. We won't waste time here by repeating such details and besides, to cover every C compiler would be an impossibility.

Following on from this introduction, the reader will now be taken on a programming course exploring some of the most advanced techniques possible with C. All the features demonstrated are illustrated with numerous programming examples - they are there to be used, altered, added to and subtracted from to meet your own requirements, so please feel free to mutilate them as much as you like. Finally, if you are new to C, do not be overwhelmed by the initial sight of what looks like a complex instruction manual; possibly the hardest part of C is deciding to spend your hard earned money on a compiler – the rest is easy!

CHAPTER 2

HARDWARE AND SOFTWARE

2.1 Introduction

Before studying the particular capabilities of your C compiler, it is both interesting and important for C programming to fully understand the basic functions and components of your computer system. We shall see briefly which features are common to all systems – whether they cost several thousands of pounds or less than a hi-fi system. You are recommended to use the rest of this chapter for reference; any reader requiring comprehensive information on the architecture of computer systems should refer to one of the numerous books on the subject.

2.2 Hardware Components

All computer systems have several components of *hardware* in common, notably the memory, the I/O devices and the central processing unit. The rest of this section gives further details about this hardware.

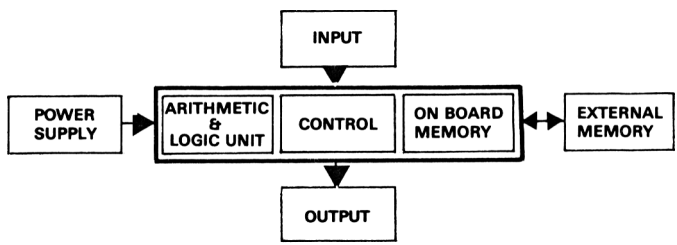


Figure 2.1 Principle components of a computer system.

The Central Processor:

The CPU (or *Central Processing Unit*) is responsible for controlling all the operations of the computer. It is also used to evaluate mathematical expressions involving addition, subtraction, multiplication and division (arithmetic), and to test if numbers are positive, negative or zero (logic). Whilst the requirements of the arithmetic functions in our C programs are obvious, the logic function is needed so that the computer can make decisions about what action to perform next.

I/O Devices:

Data is entered into and accessed from the computer by means of input and output (I/O) devices. Examples for small microcomputers include the keyboard and joysticks for input, the screen and loudspeaker for output, and a cassette unit or disk drive for both input and output. We shall see later that all I/O from C is done via *logical files* and that includes not only what we usually think of as files (i.e. on tape or floppy disk) but also the I/O devices mentioned above. With some microcomputers, such as the Amstrad and the ZX Spectrum, the logical files are referred to as *streams*. The idea of logical files is quite common in computing although initially it can be a difficult concept to grasp. Fortunately for the C beginner, the files for I/O to the keyboard and screen are handled automatically within the library functions.

Memory:

The hardware unit whose function is to store all the data held in the system is called the *memory*. The best analogy to a computer's memory is the visualisation of a long sequence of consecutively numbered storage boxes, or cells, each identified by its unique number (imagine the lockers at a railway station). The label of each storage box (or cell) in the memory is called its *address*, and each box is capable of storing one number. With many microcomputers, including the Amstrad and the ZX Spectrum, this number must be in the range 0 to 255.

Note–whilst it is possible for the BASIC programmer to work in isolation from the ideas of the memory and its structure, a C programmer must fully comprehend the concepts.

Microcomputers contain two different types of memory. These are *read only memory (ROM)* and *random access memory (RAM)*, the difference being:

ROM – is memory which is manufactured pre-programmed with permanent data. The contents cannot be changed by the user.

RAM – is general purpose memory in which data may be stored and then retrieved when required. It loses its contents when the power is switched off.

In addition to numbers, it is possible to store characters and program instructions in the memory by representing them as numerical values; these values are known as character codes and instruction codes.

Apart from storing a users program and its associated data, one section of the memory is reserved for system software. System software is a program that is stored permanently in the computer and is used for controlling all the operations of the computer system. It must be protected from corruption by the programmer, and must not be affected by switching off the power supply. A computer manufacturer always stores system software in ROM.

One final comment about memory – for the time being – is on quantities of storage capacity. When referring to a size of memory the abbreviation *K* is frequently chosen to represent 1024 memory cells. 1024 is 2 raised to the power of 10, so, an 8K program would require 8 x 1024 cells of memory.

2.3 Types of Software and how they work.

Having taken a quick browse through the basic hardware that all computer systems contain, we shall now examine the different types of software and see how the computer copes with them.

It was pointed out earlier that the cells in the computer's memory could store programs by representing the instructions as unique numerical values; for example, the code to add two values together with a particular CPU might be 35. Other codes would be used for other operations. Since a program would need to contain more than one instruction for it to be of any practical use, such a program might look like this:

35 27 133 211 23 39 5 221 100 . . .

Obviously, this program is meaningless to anyone who has not been told what instructions each value represents. This type of program is known a machine code program and although it may seem tedious and difficult to use, computer programmers in the days of the first computers had no alternatives to this type of programming. It soon became obvious to the programmers that it would be more efficient for them to use codes that bore some resemblence to the instructions that they represent; but, at this stage it is important for the reader to realise that a CPU can only ever understand programs that are in the machine

code format. Thus if a program is written using other instruction codes, it must at some stage be translated into the equivalent machine code program. This translation is accomplished by yet another computer program – ultimately written in machine code.

The next stage in the advancement of software was the introduction of assembler programs. These enable the programmer to replace machine code instructions like 35 or 169 by abbreviations for the instruction that the code represents; for example, LD for Load, ADD for Addition, SUB for Subtraction, etc. It takes no imagination to realise that an assembler program is far easier to write, read and comprehend than a machine code program. These abbreviations, which are commonly referred to as *mnemonics*, are then converted into their corresponding machine code values by a section of software called an *assembler*, and then the machine code version is used as before. As long as the machine code works, the computer does not care where it comes from.

A futher advance in software since the late 1950s has been the development of several sophisticated programming languages known as high level languages. There are many different ones, the most common being Fortran, ALGOL, COBOL, Pascal, CORAL, BASIC and, of course, C. The choice of which language to use depends on the application. For example, a large data processing department would probably use COBOL, a scientific application might use Fortran and an application that required results to be produced within strict time limits could well use CORAL. Modern languages, such as C, are very flexible and not really limited to any particular area of application. Unlike assembler programs, the codes in high level languages do not correspond one to one with the machine code values but, instead, allow the coding to resemble the nature of the problem to be solved. This can be seen below in a small C program which prints the numbers 0 to 9.

Program 1: counter

```
main()
    {
    int i;

    for (i = 0; i < 10; i++)
        printf("%d \n",i);
    }
```

Whilst the reader should not, at this stage, try to understand the C syntax, it should be evident that a high level language, when compared with the other types of software, is by far the easiest for readability. As with assemblers, before a CPU can understand the codes of a high level language, they must be translated into their machine code equivalent by a special program. It might be reassuring to know that you are totally isolated from this translation process – you do not need to know how the process works, just that it does. All we shall say on the subject is that there are two methods used for translating high level languages into machine code versions. The first method, which is used by most implementations of BASIC on microcomputers, uses software called an *interpreter* to convert each small part of the high level language program into machine code as it is needed. This means that if a section of code is executed several times, then it must be translated several times and so will slow down the overall process. The alternative method used by many languages including C, requires software called a *compiler* which converts the whole program into machine code once and for all and then the machine code version is used every time. Since C is compiled it can produce the fast action programs not available from interpreted BASIC.

CHAPTER 3

THE DESIGN & DEVELOPMENT OF C PROGRAMS

3.1 Introduction

We can now to examine the various stages in the development of a C program. A beginner to C should concentrate more on learning the skills of structured program design, which are completely different from those used in producing BASIC programs, than worrying about mastering all the features of C at once. Even an experienced C programmer would not get straight down to the coding level and expect to produce a program that functioned correctly first time, and which contained no logical errors.

3.2 Program Development.

C program development may be split up into four distinct stages.

The first stage is for the programmer to recognise the problem to be solved and ensure that it has been correctly defined. This may seem rather trivial but system consultants frequently find that their clients are not totally sure of their requirements. The most obvious method of tackling a complex problem is to break it up into smaller sub-sections which can be considered independently as separate problems. This also enables errors and difficulties with the complex problem to be easily located.

In the second stage the programmer has to recognise the factors in the problem that are liable to vary (know as the *variables*), and then find the mathematical relationships that govern them. By using his skills the programmer can analyse this information to produce a series of rules that solve the problem; such a series is called an *algorithm*. C design skills can ideally be learned by studying good quality software written in books and magazines by C experts.

Provided that enough thought goes into these first two stages, the programmer should find the third stage reasonably simple. This involves turning the algorithm into a C program and compiling it into a machine code version that the computer can execute. Care should be taken not to introduce errors in the program; syntax errors will be detected during compilation, but logical errors cannot be discovered and will leave bugs in the compiled machine code.

The final stage is for the programmer to test the program to ensure that it functions as intended. Through testing a programmer may detect errors but unless the program is tested using all feasible input data, and through all possible paths in the program flow, the absence of errors cannot be guaranteed. Taking, for example, just the variability in numeric input, the range of test data is so vast that total testing is virtually impossible. Thus, a programmer should choose selected data that will take all possible paths in the program flow.

3.3 Structured Design.

The end result in the design of an algorithm will be a series of instructions that are obeyed in a specific order, this normally being the sequence in which they are written. It is, however, possible for the algorithm to include decisions and jumps which cause the execution of the algorithm to proceed with different instructions. In order to cope with the program flow in complex algorithms that contain numerous branches, some programmers like to draw diagrams called *flowcharts*. Flowcharts contain the instructions of the algorithm written within symbols of differing shapes, and are linked together by arrows or *flowlines* to indicate the sequence of operations. The type of symbol drawn depends on the type of instruction it represents.

Having said that some programmers like flowcharts, the majority do not! The two dimensional nature of paper actually encourages careless design, leading to inefficient programs with logic that is incomprehensible to anyone except the original author. The design is also hampered from the beginning by a preoccupation with detail – a so called *bottom-up* methodology.

Since design is so vital in program development, programmers have turned to better methods. These methods are generally called *top-down* but have assumed other names including *structured programming*, *modular programming* or the *Jackson Method*. Each of these involves structuring the problem by breaking it into simpler sub-problems and, ultimately, into very simple "building blocks".

With structured techniques and the use of careful programming, it is possible to avoid the GOTO statement. This statement permits the forward and backward branching in a program and, therefore, is the most frequent cause of lack of

structure and spaghetti tendancies. The object of removing these jumps is to produce a program comprised of a linear sequence of building blocks, as illustrated in Figure 3.1. Each block can be formally delimited by BEGIN and END.

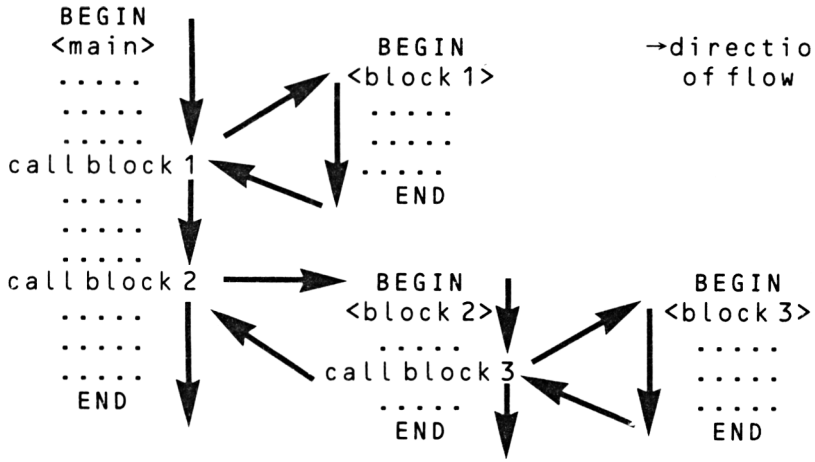


Figure 3.1 Structured design.

The top-down approach can now be seen. The main block is designed and implemented with the calls to the other blocks inserted (these can initially return immediately to the main block). The lower level blocks can then be designed and programmed independently and introduced to the higher level blocks when complete. If required, the lower level blocks can be called from more than one position.

The blocks can be designed with English statements using a simple program description language (PDL) to describe their operation. Briefly, here is a summary of PDL.

Each sub unit of a block is one of the following statement types:

(a) the simple sequence – a series of actions

(b) the alternative clause – `if` (conditional statement)
`then` (action)
`else` (other action)

(c) the choice unit

```
– case of
    1: (action 1)
    2: (action 2)
    ....
    n: (action n)
  end case
```

(d) the iteration unit

```
– while (condtional statement)
    do (action)
    end do
```

(e) the repetition unit

```
– repeat
    (action)
  until (conditional statement)
```

Standard actions, such as I/O, may be included in the description.

As an example of PDL, consider the problem of solving the quadratic equation,

$$ax^2 + bx + c = 0$$

where a, b and c are know constants and x is required to be found. It can be proved mathematically that the solution is

$$x = \frac{-b + \sqrt{b^2 - 4.a.c}}{2.a}$$

```
function quadratic (a, b, c)
  begin
    if a = 0
    then
      begin
        if b = 0
        then print "No solution"
        else print 'Linear equation x = "; –c/b
      end
    else
      begin
        d = b² –4ac
        if d > 0
        then print "x1 =";(–b + √d)/2a;
                   "x2 =";(–b – √d)/2a
        else print "No real solutions"
      end
```

Once the PDL has been properly designed, it is a simple task to translate it into a high level language such as C, and be reasonably confident that it will work correctly the first time.

It is not within the scope of this book to give a full exposition of PDL but its use is recommended for C program design. Detailed information and examples on the subject can be found in "Successful Software for Small Computers" by Graham Beech, published by Sigma Press.

3.4 A Brief Look at C Programming.

As we shall soon see, the structure of C closely resembles that of PDL. A C program, whatever its size, consists of one or more functions (analogous to a block of PDL). Each function is distinguished by a unique name and control may invoke another function simply by calling its name.

Program 2: main

```
main()
    {
    }
```

Program 2 is the simplest C program in the world consisting of a single function called `main` and not containing any executable statements. All programs must have one function called `main` which is the first one performed when a program is executed; if `main` is missing then the program will not compile. Program 2 example also illustrates that every function begins and ends with braces `{` and `}`. In C, each statement is followed by a semicolon, and the braces are also used to group statements together into blocks, or *compound statements*, which are treated by the C compiler as equivalent to single statements.

Program 3: print

```
main
    {
    printf("SIGMA PRESS");
    }
```

It is very unlikely that you will ever write a program containing just one function. Even the simplest programs have to call other functions to output results (usually standard I/O library functions, e.g. `printf`). Program 3 also illustrates passing parameters to other functions (i.e. the sequence of characters

"SIGMA PRESS"). This is a mechanism whereby procedures can communicate with each other which we shall study in detail in Chapter 5.

A program may be built up of functions contained in more than one source file. This enables functions of a specific type (e.g. I/O, database, etc.) to be stored together and be used as required. Some compilers are supplied with sophisticated linking utilities enabling files to be compiled (even without a `main` function) into *relocatable* machine code modules. When a program is compiled, any specified libraries are searched for missing functions. The alternative method is to merge source files into the file being compiled; this is done using the *preprocessor command* `#include` which specifies a source file to be included in compilation.

Example: `#include <dbase.c>`

A preprocessor command is one preceded by the # sign. This part of C is not clearly defined since different systems have different requirements, and so you should refer to the documentation supplied with your compiler for a full description.

The Hisoft C compiler, which does not have any linking facilities, contains a special form of the `#include` command giving a conditional compilation facility. If the specified filename is enclosed within question marks, then only functions which have been invoked previously are included for compilation. This command would normally be the last instruction in the file. When constructing a Hisoft library file, it is important to order the functions such that those which are invoked by other functions appear first in the file.

Example:

```
main()
    {
    .....
    (actions)
    ....
    }
#include ?common.lib?
```

Comments may be included in C programs to assist understanding, but it is not necessary to overburden the source with excessive comments since it is usually built up of many small functions each doing a simple task. If you understand what action the function has, then grasping the overall operation should be simple. Ideally, the function names should be descriptive of the function tasks. Comments that are required may be placed anywhere and are enclosed by `/*` and `*/`.

3.5 Using Hisoft C.

Since operating the Hisoft compiler is a little unorthodox, we shall take a brief look at its use. Once loaded, the following sign-on message is displayed:

```
HISOFT-C Compiler V1.2
Copyright © 1984 HISOFT

>
```

The > prompt sign means the computer is in "edit" mode and a C program may be entered using the Hisoft line editor. When completed, the program can be compiled. To switch to "compilation" mode enter "C" and the screen will first clear and redisplay the sign-on message; this time the > prompt will not be present. The C source may then be compiled using the `#include` command; if no filename is specified, then the C source stored in the editor is included and compiled. When all files have been included, hit the End of File key (CTRL Z on the Amstrad, SYMBOL SHIFT I on the ZX Spectrum). Provided that no errors have occurred, the compiler will display:

```
Type y to run program:
```

The program may then be re-run any number of times by typing "y".

3.6 And so ...

There is much more to learn about C programs such as how to control program flow and manipulate data, but we shall leave these topics to later in the book. We shall now take a detailed look at how the data is stored in the computer, since such knowledge is imperative for serious C programmers.

CHAPTER 4
DEEP C

4.1 Introduction

Some people like to compare the memory in a computer system with that of the human brain; however this comparison leaves a lot to be desired. The brain can memorise all types of information including letters, numbers, words, pictures, etc. In comparison, the computer's memory is very simple. The nature of electronics restricts components to being either switched on or off and this reduces the computer's capabilities to the recognition of just two states, which are written for convenience as "0" and "1". Just as in English where words and sentences are built up by using more than one letter and numbers consist of several digits, computer expressions are represented by sequences of 0s and 1s. Such patterns of 0s and 1s are called *binary numbers*.

4.2. Binary Numbers

When numbers are written in the normal decimal or base 10 representation, the digit furthest to the right gives the number of units, the digit to its left gives the number of tens (the base of the number system), the next digit to the left gives the number of hundreds (the base squared), and so on.

Binary numbers use a base of 2; the digit furthest to the right gives the number of units, the digit to its left gives the number of twos (the base), the next digit gives the number of fours (the base squared), the next digit gives the number of eights (the base cubed), and so on.

Conventionally, the number 345 in the familiar decimal numbering system that we use means:

```
3   4   5   (decimal)
|   |   |
|   |   |____ 5 x 10^0 = 5 x   1 =   5
|   |________ 4 x 10^1 = 4 x  10 =  40
|____________ 3 x 10^2 = 3 x 100 = 300
                                   345 (decimal)
```

Likewise a binary number such as 11001 is equivalent to

```
1   1   0   0   1   (binary)
|   |   |   |   |
|   |   |   |   |____1 x 2^0 = 1 x  1 =  1
|   |   |   |________ 0 x 2^1 = 0 x  2 =  0
|   |   |____________ 0 x 2^2 = 0 x  4 =  0
|   |________________ 1 x 2^3 = 1 x  8 =  8
|____________________1 x 2^4 = 1 x 16 = 16
                                       25 (decimal)
```

When counting, a decimal "1" is carried over into the next column whenever a "9' is reached, i.e. after 9 comes 10, after 29 comes 30, after 99 comes 100. Similarly, since binary only uses the digits 0 and 1, a "1" is carried over whenever a "1" is reached, i.e. after 0 comes 1, after 1 comes 10, after 10 comes 11, after 11 comes 100 (Further examples of binary numbers are given later in Figure 4.4).

It can be seen that the binary numbering system works on the same principle as the decimal numbering system, but since more digits are required to represent a number in binary than in decimal it is more cumbersome for us humans to use.

It is now possible to explain why, in Chapter 2, it was stated that many small computers can only store numbers between 0 and 255 at any particular memory address. This is because a storage location can only contain eight binary digits; thus the range is from 00000000 (0 decimal) to 11111111 (255 decimal). When referring to binary numbers in computers an individual digit is called a *bit* and a group of eight bits at an address is called a *byte*.

Binary arithmetic

Binary numbers may be combined by addition, subtraction, multiplication and division.

Binary addition uses the following five rules:

```
        0 + 0 = 0
        0 + 1 = 1
        1 + 0 = 1
        1 + 1 = 0 carry 1
carry + 1 + 1 = 1 carry 1
```

example:

```
  1 1 1   Taking the 1st column,      1 + 1          = 0 carry 1
+ 0 1 1   Taking the 2nd column,      1 + 1 + carry  = 1 carry 1
-------   Taking the 3rd column,      1 + 0 + carry  = 0 carry 1
  1 0 1 0 Taking the 4th column,                carry = 1
-------
1 1 1
```

Carry indicators

Binary subtraction uses the following four rules:

```
0 - 0 = 0
1 - 0 = 1
1 - 1 = 0
0 - 1 = 1        (found by borrowing from the next higher digit)
```

example:

```
  0 1   Taking the 1st column,    0 - 1 not possible borrow
  1 0                             from next higher digit,
- 0 1                             10 - 1 = 0
-----   Taking the 3rd column,    0 - 0 = 0
  0 1
-----
```

Example:

```
  0 1 1   ←+
  1 0 0
- 0 0 1
-------   Borrow
  0 1 1   indicators
-------
```

Example:

```
  0 1    ←+
  1 0 1
 -0 1 1
 -------
  0 1 0
 -------
```

One complication when dealing with binary numbers is the handling of negative numbers. Consider the problem of subtracting 92 from 20 using binary numbers:

Decimal	Binary
20	0 0 0 1 0 1 0 0
−92	0 1 0 1 1 1 0 0
−72	1 0 1 1 1 0 0 0

The above binary result is obtained only if we do not worry where the "borrow" from the left-hand bit came from. But, 10111000 in binary is equal to 184 in decimal, so that −72 in decimal seems to have the same binary representation as 184. How can we explain this? To do so, we must find a way of representing negative numbers. These can be visualised as numbers in the opposite direction to postive numbers, i.e. if an electronic meter reading 0000 was wound back one unit it would read 9999. To obtain −72 it should be wound back 72 units, as shown in Figure 4.1.

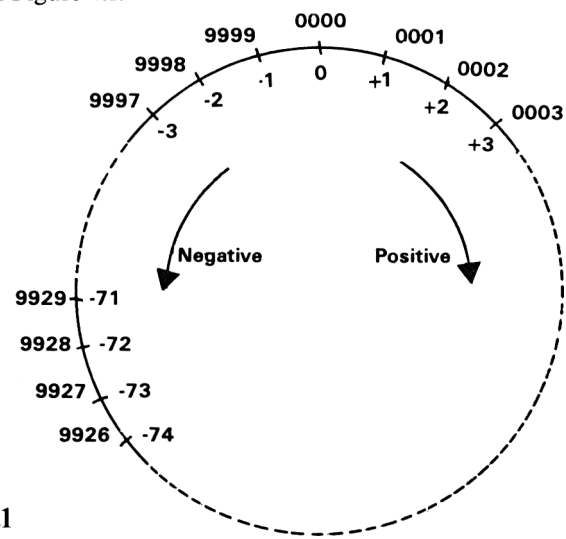


Figure 4.1

Thus −72 can be represented on our meter as 9928 - as long as we remember that it is really a negative number.

Using a similar method for binary numbers, it is convenient to represent negative numbers in a form that is known as the *two's comlement* form. In a binary number containing x bits, this simply means that minus n is stored as $(2^x - n)$.

For example, a computer using 8 bits would store −72 (decimal) as $(2^8 - 72) = 184$ (see Figure 4.2).

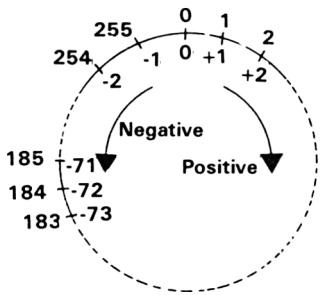


Figure 4.2

A binary number can be converted into its two's complement form simply by converting all the 1s to 0s, and the 0s to 1s, and then adding 1.

Example:

negate 72	− 72 in binary is	0 1 0 0 1 0 0 0
	Convert digits	1 0 1 1 0 1 1 1
	Add 1	1 0 1 1 1 0 0 0

So −72 (decimal) is represented by 10111000

Example:

negate (−72)	− (−72) in binary is	1 0 1 1 1 0 0 0 (from above)
	Convert digits	0 1 0 0 0 1 1 1
	Add 1	0 1 0 0 1 0 0 0

So $-(-72)$ (decimal) is represented by 01001000, which, as we would expect, is 72 (decimal).

If this is valid we would also expect $72 + (-72)$ to be zero.

$$\begin{array}{r} 01001000 \\ +\ 10111000 \\ \hline 1\ \ 00000000 \end{array}$$

Ignoring the first digit, the last eight bits show this to be true. The additional bit on the extreme left is known as the *sign bit*. If the sign bit is "0" the number is postive and if the sign bit is "1" the number is negative. In this case, where the numbers add up to zero, the sign bit is irrelevant. The use of the sign bit simply enables the computer to distinguish between positive and negative numbers. As further examples, we would expect $172 + (-25)$ to be postive and $117 + (-159)$ to be negative.

Decimal	sign	Binary	
172	0	10101100	
− 25	+ 1	11100111	← −25 in two's complement, i.e. sign bit set
147	0	10010011	

positive, so result is in normal form

Decimal	sign	Binary	
117	0	01110101	
−159	+ 1	01100001	← −159 in two's complement, i.e. sign bit set
− 42	1	11010110	

negative, so result is in two's complement form

4.3 Hexadecimal Numbers

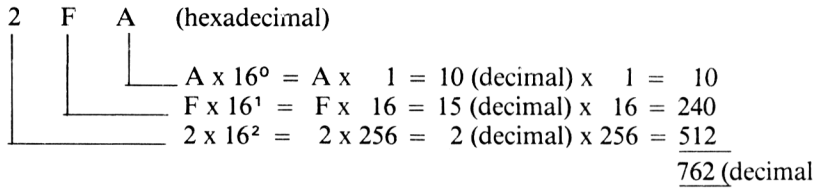
Unlike computers, we find these long strings of 0s and 1s difficult to memorise and awkward to work with, so some simpler notations have been developed to help us. One such system uses a numbering notation with a base of sixteen and

is known as the hexadecimal system. This requires sixteen digits, but instead of designing six new symbols the first letters of the alphabet are used. The sixteen hexadecimal digits are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

It is important, in this context not to interpret the new symbols as letters, but as digits that are capable of the mathematical operations such as addition, subtraction, etc.

As an example, consider the hexadecimal number 2FA:



There is a very simple method for converting binary numbers into their hexadecimal equivalent. This involves splitting up the binary digits into groups of four bits, starting from the right. Each digit of the hexadecimal number is then represented by the value of the four bits.

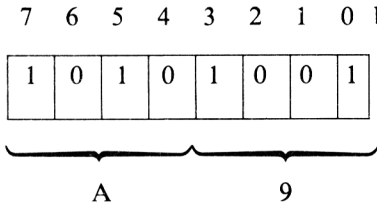
Example:

	10	1111	1010	
Split into groups of four bits	10	1111	1010	
Calculate value of each group	2	15	10	(decimal)
Convert to hexadecimal	2	F	A	(hexadecimal)

Thus 2FA (hexadecimal) and 1011111010 (binary) are equivalent.

As a byte contains eight bits, its value may be expressed with just two hexadecimal digits; thus, with the possible exception of decimal numbers, the hexadecimal numbering system is often used in computing in preference to any other.

Example:



The first hexadecimal digit represents the value of the four most significant bits and the second represents the value of the four least significant bits.
So the hex value of this byte is A9.

Figure 4.3

4.4 Octal Numbers

One other important number system required by C programmers is Octal which uses a base of eight. This is used in C for character/string constants to drive graphics, sound, etc. Examples of octal and other notations is given in Figure 4.4.

Decimal	Binary	Hex	Octal	Decimal	Binary	Hex	Octal
-15	11110001	F1	161	1	1	1	1
-14	11110010	F2	162	2	10	2	2
-13	11110011	F3	163	3	11	3	3
-12	11110100	F4	164	4	100	4	4
-11	11110101	F5	165	5	101	5	5
-10	11110110	F6	166	6	110	6	6
- 9	11110111	F7	167	7	111	7	7
- 8	11111000	F8	170	8	1000	8	10
- 7	11111001	F9	171	9	1001	9	11
- 6	11111010	FA	172	10	1010	A	12
- 5	11111011	FB	173	11	1011	B	13
- 4	11111100	FC	174	12	1100	C	14
- 3	11111101	FD	175	13	1101	D	15
- 2	11111110	FE	176	14	1110	E	16
- 1	11111111	FF	177	15	1111	F	17
0	00000000	0	0	16	10000	10	20

Figure 4.4 Examples of decimal, binary, hexadecimal and numbers.

Of course, there are many other numbering systems using bases of 3, 4, 5, etc, however to program in C it is sufficient to just use the notations that we have just studied.

When working with decimal, hexadecimal and octal numbers you should remember that although the computer appears to understand them through its I/O devices, it still stores and operates on them in their expanded binary form.

Although these strange notations may seem difficult at first, once they are used in practice, understanding them becomes second nature. A full understanding of these numbers is vital to be able to use the advanced techniques that are available in C programming.

4.5 Computer Logic

If we consider the normal every day things which we do, we find that the majority of these tasks require the making of numerous decisions. For a computer system to simulate some of the mental processes of the human brain, it must also be able to make decisions. Although a computer can only think in terms of 0s and 1s, it is still possible for the computer to solve complicated tasks involving decisions by reducing the problems to ones of 0s and 1s. This is *binary logic* and is often referred to by mathematicians as *boolean algebra.*

This logic can be explained most clearly by considering the following example:

```
if my car was serviced more than 6 months ago                    (A)
and I have driven at least 6000 miles since the last oil change  (B)
then I shall change the oil                                      (C)
```

There are three sections to this statement which have been labelled A, B and C. A and B refer to the two questions I have to ask myself and C refers to the action I shall take. A and B are sometimes called binary variables, and take the value "1" if the statement is True and "0" if it is False. Analysing the statements reveals that there are four possible combinations for A and B which are shown below.

1. Car has been serviced within 6 months.
   Less than 6000 miles have been driven.
2. Car has been serviced within 6 months.
   More than 6000 miles have been driven.
3. Car has not been serviced within 6 months.
   Less than 6000 miles have been driven.
4. Car has not been serviced within 6 months.
   More than 6000 miles have been driven.

Since I only change the oil if statement A `and` statement B are True, then I only

change it for the fourth option, i.e. if the car has not been serviced within 6 months and more than 6000 miles have been driven since the last oil change.

These results can be written in tabular form using 1 for True and 0 for False. Such a table is called a *truth table*, an example of which is shown in Figure 4.5.

AND	A	B	C
(1)	0	0	0
(2)	0	1	0
(3)	1	0	0
(4)	1	1	1

Figure 4.5 The AND truth table.

If, however, the original statement read:

```
if my car was serviced more than 6 months ago                    (A)
or I have driven at least 6000 miles since the last oil change   (B)
then I shall change the oil                                      (C)
```

Then I would change the oil if either (or both) statement A or statement B was True. The truth table would be as shown in Figure 4.6.

OR	A	B	C
(1)	0	0	0
(2)	0	1	1
(3)	1	0	1
(4)	1	1	1

Figure 4.6 The OR truth table.

4.5 Logical Operators

It is possible to combine conditional statements in the English language by linking them with either `and` and `or`; similarly, you can it is also possible to combine conditional statements in PDL and the C language by using operators. These operators are called *AND* and *OR*, although in the C source they are written as `&&` and `||` respectively. These operators, which are very important in decision statements (they appear in three of the PDL sub units which we met in Chapter 3), are used between conditional statements as shown.

(condition 1) && (condition 2)	AND returns a result True if, and only if, both conditions are True. Otherwise the result is False.
(condition 1) || (condition 2)	OR returns a result False if, and only if, both conditions are False. Otherwise the result is True.

These results are summarised in the following tables:

AND	True	False
True	True	False
False	False	False

Figure 4.7

OR	True	False
True	True	True
False	True	False

Figure 4.8

There is one other logical operator, *NOT* that is used in conditional statements. In C, this is written as !.

! (condition 1)	NOT returns a result True if the condition is False and a result False if the condition is True

NOT	
False	True
True	False

Figure 4.9

Care has to be taken when programming in C since the && and || operators are not commutative; the evaluation of each term in a logical expression ceases as soon as it is known that the result of the expression cannot be changed.

Example: a && b

If a is False then b is not evaluated and a result False is returned. If b is True then b is evaluated to determine the result.

It is also possible to combine more than two conditional statements using more than one operator.

Example:

! (condition 1) && (condition 2) || (condition 3) && (condition 4)

However, logical operators are processed in a fixed order and not simply from left to right. First all NOTs are evaluated, then the ANDs and finally the ORs.

One other logical operator which is occasionally mentioned, but not available directly in C is *EX-OR* (exclusive OR) which operates on two operands. This gives the following results.

(condition 1) EX-OR (condition 2)

EX-OR returns a result True if one, and
only one of the conditions is True.
Otherwise the result is False

EX-OR	True	False
True	False	True
False	True	False

Figure 4.10

One final comment about logical operators in C. Up to now we have defined True as a value of 1. In fact, the C compiler considers any non zero value as True, and a zero value as False. C condition statments may consist of two algebraic expressions or two characters separated by one of the following:

== ... equal to

< ... less than
<= ... less than or equal
> ... greater than
>= ... greater than or equal
!= ... not equal

4.7 Bitwise operators

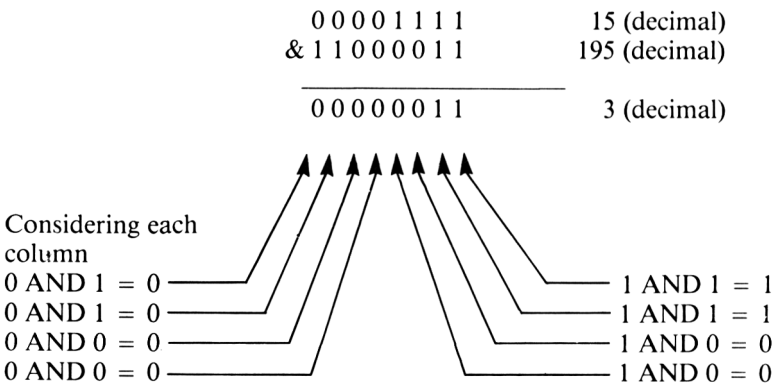
C provides a number of operators for bit manipulation. Some of these are similar to the logical functions that we have just met, but they work on separate bits rather than byte values 0 and 1.

The bitwise operators are:

&	bitwise AND
|	bitwise OR
^	bitwise EX-OR
<<	left shift
>>	right shift
~	one's complement

Examples:

15 & 195 returns 3. How can we explain this ? To do so we must convert 15 and 195 into binary numbers and then use our AND truth table on each column.



Likewise, if the bitwise OR or EX-OR operators are used on two numbers, each column is evaluated using the corresponding truth table. The one's complement operator converts all the 0s in the binary form to 1s, and vice versa.

The shift operators << and >> perform left and right shifts on the binary form for the specified number of bit positions.

Example:

10 << 1 returns 20

(i.e. a shift of 1 is equivalent to multiplying a number by 2. This is similar to multiplying a decimal number by 10 by just adding a zero to the last column. 99 x 10 = 990.)

10 (decimal) = 00001010

Shift by 1 place 00010100 = 20 (decimal)
(fill vacated bits with 0)

With bitwise operators we have a mechanism for setting and determining the value of specific bits within a byte. This involves a technique called *masking*.

Examples:

1. To select a single bit X.

AND the byte with another containing just a single bit set – every bit except X is set to zero.

```
1 1 0 1 X 1 0 1
0 0 0 0 1 0 0 0
---------------
0 0 0 0 X 0 0 0
        ▲
        X AND 1 = X
```

So, if the result is 00001000 (8 decimal) then bit 4 (X) must have been set.

2. To unset a single bit X.

AND the byte with another which has all bits set except the one corresponding to X – only the required bit will be altered.

```
1 1 0 0 0 1 X 0
1 1 1 1 1 1 0 1
---------------
1 1 0 0 0 1 0 0
          ▲
          X AND 0 = 0
```

3. To set a single bit X.

OR the byte with another which has all bits zero except the one corresponding to X – only the required bit is altered.

i.e.

```
Y Y Y Y X Y Y Y
0 0 0 0 1 0 0 0
---------------
Y Y Y Y 1 Y Y Y
        ↑
    X OR 1 = 1
```

4.8 Stop

Before proceeding further, it is important that you understand the principles of this chapter, as the concepts will be commonly used throughout the book.

CHAPTER 5
DATA TYPES

5.1 Introduction

While everybody realises that a computer can handle very complicated arithmetic expressions, it is a common misconception that this is a simple task – in fact it is one of the hardest! There are two main reasons for this. The first being that a computer is expected to be able to cope with a vast range of numbers and the second is that an individual address in the computer's memory can only contain whole numbers between 0 and 255 inclusive. It is, however, comforting to know that when a high level language such as C is used, the programmer needs no knowledge of the complicated methods used to evaluate arithmetic expressions; these are handled automatically by the compiler.

5.2 More on Numbering Notations

Those readers who have little experience of mathematics have probably never given a great deal of thought to the many notations in which numbers can be written. For example, the first three different types of numbers we ever learned about were whole numbers, fractions and decimals. Computing, similarly uses three types of numbers – integers, real numbers and exponential numbers. We will now consider these in more detail.

Integers.

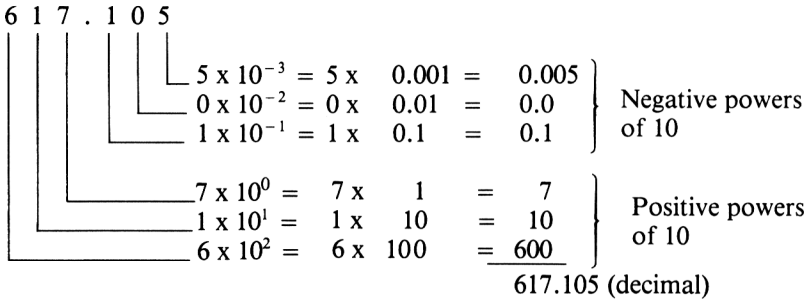
Integers, commonly called whole numbers, are numbers comprised of a sequence of digits which does not involve a decimal point or any fractional component. They may be positive or negative. For example, −156, −22, 0, 1, 2 and 69 are all integers.

Real Numbers.

Real numbers are a sequence of decimal digits with a single decimal point either at the end, the beginning or between any two digits. The digits to the left of the decimal point are weighted in positive powers of 10 and form the integer

component of the number. Likewise the digits to the right of the decimal point are weighted in negative powers of 10 and form the fractional component of the number. Real numbers may be positive or negative. Examples of real numbers include -2.3, 0.4, 21.035, .25, -0.278.

The following diagram illustrates this idea:



Exponential Numbers.

Exponential numbers are written in a notation which is suitable for either very large or very small numbers. They are written using a real number followed by the symbol "E" and an integer number. The "E" means *times 10 to the power of.*

Thus yEx means y times 10 to the power of x. Further examples are given by:

$$1.234E-2 = 1.234 \times 10^{-2} = 0.01234$$
$$1.234E0 = 1.234 \times 10^{0} = 1.234$$
$$1.234E2 = 1.234 \times 10^{2} = 123.4$$

The effect of the integer following the "E"is to indicate how many places the decimal point has to be shifted. A positive value means shift to the right and a negative value means shift to the left.

For example, 1E10 means 1 followed by ten '0's, i.e. 10,000,000,000

In the current release of the Hisoft compiler, numbers are restricted to the integer notation.

The hexadecimal and octal notations, which we met in Chapter 4, may be used freely in C programs by writing the number preceded by 0x and 0 respectively.

Example:

0x4C and 0114 would both be interpreted as 76.

5.3 Variables

All items of data stored in the computer's memory are identified in C by labelling them with symbolic names. Such a name refers to the location at which the particular data is stored. If we consider a statement in which a numerical value is assigned to, for example, a name `max_val`, then the computer reserve a section of memory for this name and stores the value there. From then on, any reference to `max_val` relates directly to the value stored in that section of memory specifically reserved for `max_val`. It is because the data stored at the location may be changed that the names are called *variables.*

In C, there are certain rules for choosing the names of variables. They can have names of any length and may consist of letters, digits and the underscore '_', character provided that the name starts with a letter or '_'. However, C variables are only recognised up to the eighth character. Thus `position_1` and `position_2` refer to the same variable (i.e. `position`).

Naturally, in order to improve the readability of a program, it makes sense for the variable names to be chosen so that they bear some resemblence to their contents.

5.4 Declarations

All variables must be declared before they are used. If they are declared at the top of the C program they are *global* and are available for use from anywhere in the program. *Local* variables are declared at the top of blocks of code, before any executable statements, and may only be used from within that block. Local variables in different blocks may have the same name but they are totally independent.

Local variables may be declared as *automatic*, *static* or *register*. Automatic variables are discarded on exit from the block, whereas static variables retain their values until the block is called again. Register variables are treated like automatic variables but are stored in locations in the CPU (if available), and so can speed up the processing in sections of code where they are frequently used. *External* variables may be declared and refer to global variables that have been declared elsewhere – possibly in another source file.

Examples:

```
int average;
int a, b, c;
int ptr = 0;
static int max;
register int count;
extern char val_ptr;
```

Declares integer variable (automatic).
More than one variable may be declared at once.
Variables may be *initialised* when declared.
Static variable.
If no registers are available, then the variable is treated as a normal automatic variable.
Informs the compiler that the variable has been declared elsewhere.

In order to keep their compiler small, Hisoft have imposed several minor restrictions in their version of C. Automatic variables cannot be initialised when declared; instead, values must be assigned to them. Because there are never enough CPU registers free, the `register` keyword is accepted but ignored. And finally, all local variables must be declared at the start of a function and not at the top of a compound block; this restriction does, in fact, aid the understanding of programs.

The size or precision of variables is dependent upon the hardware and your C compiler, but an integer is usually stored within sixteen bits, restricting numbers to between −32768 and 32767. Integer variables may be declared as *unsigned*, in which the range is between 0 and 65535.

Example:

```
unsigned int min_val;
unsigned max_val;
```

The word int may be omitted.

In addition to `int`, C has several other data types, although the way these are handled varies, once again, with the hardware and C compiler. The complete list of data types is:

		Typical Precision
`int`	Integer	16 bits
`short`	Short integer	16 bits
`long`	Long integer	32 bits
`char`	Character	8 bits
`float`	Floating point / real	32 bits
`double`	Double precison floating point	64 bits

Long constants are distinguished by following the number with an L.

Example:

```
long start_val = 10000L;
```

The current version of the Hisoft compiler uses 16 bits for int, short and long variables, and 8 bits for char variables. Floating point numbers are not implemented. However, it is hoped that a future release will include 8 bit short variables, 32 bit long variables and floating point arithmetic.

5.5 Characters

All computers have a character set, usually of 256 items, which can be visualised as its own unique alphabet. It can include alphabetic and numeric characters, Greek letters, pixel graphics, punctuation marks, mathematical symbols, control characters, etc. Most manufacturers of microcomputers use the ASCII (American Standard Code for Information Interchange) character set, but many have altered it slightly to suit their own requirements resulting in many versions which are not really standard at all! Each character is distinguished by its own character code which, in the case of ASCII, is a unique number between 0 and 255. Since char variables comprise 8 bits they can be used to accommodate the character code of any ASCII character.

Character constants may be represented by enclosing either the character, or the character code in octal preceded by '\', within single quotation marks. Certain non-graphic control characters have a special notation represented by a backslash and a normal character:

\b Backspace
\f Form feed
\n Newline
\r Carriage return
\t Tab
\\ Backslash
\" Double quotes
\' Single quotes

Examples:

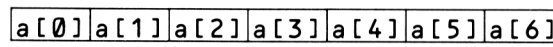
```
char a_upper = 'A';
char form_feed = '\f';
char line_feed = 10;
char escape = 0x1B;
char null_char = '\000';
```

5.6 Arrays

It is often useful to reserve a block of cells in the memory which can be referred to by a name (identifying the block) and a number (representing the particular cell within the block); such a block is called an *array*. The compiler is instructed to reserve an array by declaring it along with the other variables and specifying its length within square brackets.

Example:

```
int a[7];
```
Reserves an array, `a`, of seven integer elements.



Each cell can store one integer and is identified by the block variable name and cell number which is written within square brackets. For example, the 6th cell in the block is referred to by `a[5]`. The term in brackets is called the *subscript* and varies from 0 to one less than the specified array length. A subscript may be replaced by an algebraic expression whose result indicates the required element, but care must be taken that the result always lies in the correct range. BASIC users may be aware that if they use a subscript that is out of range, then the program stops giving an appropriate error message. C, however, allows you to use subscripts out of range, on the assumption that you know what you are doing and are deliberately trying to access a different section of the memory – thus take care!

Example:

```
int a[7], b[7];
```

Cell `a[7]` refers to the section of memory following on from the array `a`, i.e. element `b[0]`.

It is also possible to set up arrays with more than one subscript.

Example:

```
int c[3][4];
```

c[0][0]	c[0][1]	c[0][2]	c[0][3]
c[1][0]	c[1][1]	c[1][2]	c[1][3]
c[2][0]	c[2][1]	c[2][2]	c[2][3]

Such an array is called a two dimensional array; remember however, that the cells are physically stored sequentially in the computer's memory.

The same idea works for an n dimensional array.

Example:

```
int d[3][4][2][3];
```

5.7 Strings.

String constants, which are sequences of characters, are commonly required in C programs and are represented by enclosing the characters in double quotes.

Example:

```
"VOLVO"
```

One of the most common use of arrays is to store sequences of characters.

example:

`char car[6];` Could be used to store

V	O	L	V	O	\0

by assigning each character in turn i.e.

```
car[0] = 'V';
car[1] = 'O';
```

etc.

We shall see later that there are a number of standard string functions which are fundamental to string manipulation, including one which allows the address of the target array and the string constant to be specified and then undertakes to assign each character in turn. Until we meet these functions, all we shall say is that strings are terminated by a null character (i.e. the character whose character code is zero) so that the end of a string can be detected. This extra character means that the length of a string array should be 1 greater than the maximum length of the stored string. Many string functions do not check this length and so, if it is exceeded, are likely to corrupt the next area of memory.

5.8 Initialising Arrays.

Global and static arrays may be initialised by following the declaration with an element list. When the size of the array is omitted, the compiler uses the number of elements detected.

Examples:

```
int no_days[12] =
      {31,28,31,30,31,30,31,31,30,31,30,31};
int factorial[] =
                {1,2,6,24,120,720,5040,40320};
char name[] = {'J','A','N','E','\000'};
```

A shorthand for the character initialisation is

```
char company[] = "G.H.Hogg & Sons Ltd.";
```

5.9 Algebraic expressions.

When we consider mathematical problems of the form

```
5 + x / 3 - y
```

we are talking about the evaluation of *algebraic expressions*. An algebraic expression is a list of numerical variables and constants known as *operands*, linked together by *operators* such as +, − , *, etc.

Examples:

```
a * b / c,
-a + b + c - 2.5,
 a + b * c * 3
```

Brackets may also be included in such expressions in which case the algebraic expressions most deeply nested by brackets are evaluated first.

Examples:

```
(a + b + c) / 3,
(a + b) / (c + d)
```

Sometimes if we choose to omit the brackets we may not achieve the results that we require. For example, consider the expression `a + b * c`. You might think that the computer could interpret this in two ways.

1) c multiplied by the result of a plus b
2) a added to the result of b multiplied by c

Thus it is important that there are some rules to specify the order in which the operands are operated on. C gives each operator a *hierarchy* or priority value. When an expression is evaluated, all operators with the highest priority are considered and evaluated from left to right. Next, the operators with the second highest priority are evaluated, again, from left to right, an this continues until all the operators have been considered and the expression is evaluated. The list of C operators and their hierarchy is given in Figure 5.1.

To some readers, the number of C operators may seem surprising, as there are far more than any version of BASIC. There are two types of operators, *unary* and *binary* for use on one or two operands respectively.

The most common operators are + (addition), - (subtraction), * (multiplication) and / (division); these should not require any explanation.

Similar to these arithmetic operators is **%**, which returns the remainder when the first operand is divided by the second.

Example:

`a = 13 % 3` Evaluates to 1.

Adding 1 to a variable is so common that a special operator, **++**, is available to do it.

Example:

`i++` Increment the contents of i by 1.

Similarly, we can decrement a variable by using `--`.

Example:

```
j--
```

Alternatives to the above two examples are `++i` and `--j`. The difference in the two notations is only noticable when they are involved in larger expressions.

Example:

```
k = ++i ;
```
Increment i and assign result to k.

```
k = i++;
```
Assign i to k and then increment i.

When the left hand side of an expression is repeated on the right hand side, the expression may be written in a simpler notation using `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `+=` or `/=`.

Examples:

```
i = 10 * i ;
```
could be written as `i *= 10;`

```
j = j + 100;
```
could be written as `j += 100;`

```
k = k / 10;
```
could be written as `k /= 10;`

This is very useful when the expression is complicated, for example, when involving complex array subscripts, and also can produce more efficient machine code.

Example:

```
x[y[i]][z[j]] += 2;
```

The & operator returns the address of a variable, i.e. its location in the memory.

Examples:

```
ptr1 = &data;
ptr2 = &word[10];
```

In the case of arrays, the address of the first element in the array can be obtained by referencing the array without any subscript.

Example:

```
char s[10];

k1 = &s[0];
k2 = s;
```
This results in both k1 and k2 containing the start address of the array.

The remaining operators in C are the logical and bitwise operators which we met in Chapter 4.

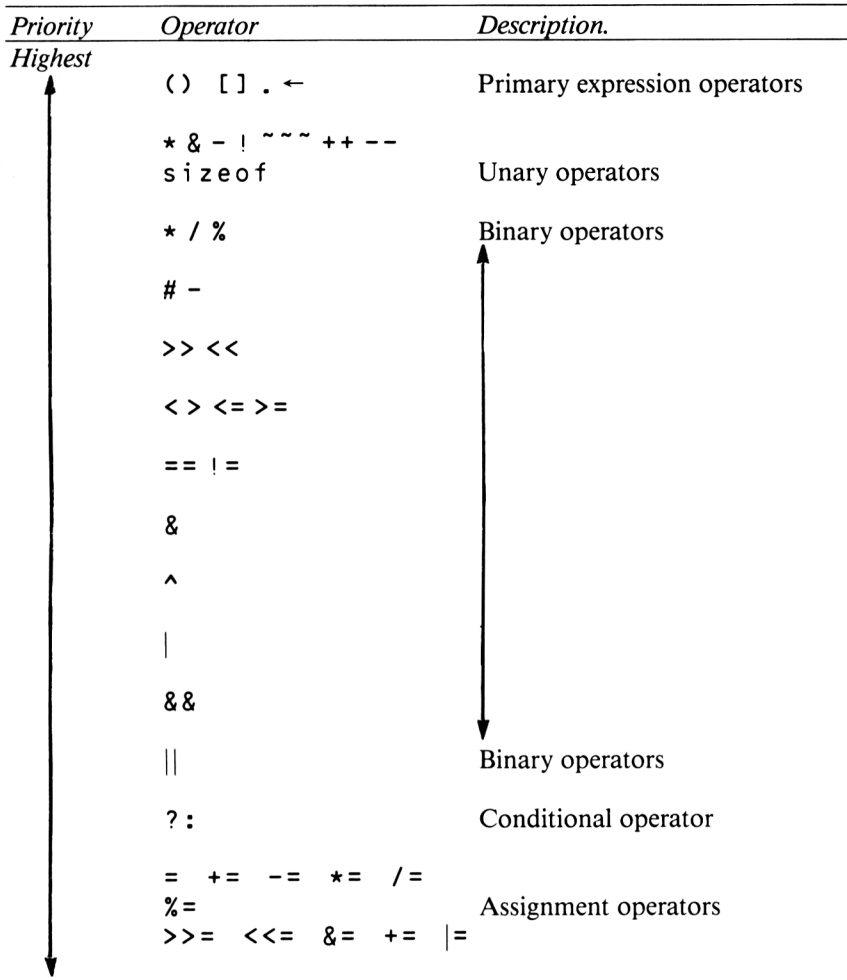


Figure 4.1 The Hierachy of C operators:

There are a couple of operators here that we have not yet studied but which have been included for future reference.

5.10 Pointers

Any reader who has not met *pointers* before may find the next concepts difficult to understand, but do perservere since their use is fundamental in C programs and it is only a poor programmer who is unable to use them constructively. Understanding pointers is not nearly as difficult as teaching someone when they should be used; however we shall meet them on numerous occasions throughout this book and, hopefully, their use will soon become second nature.

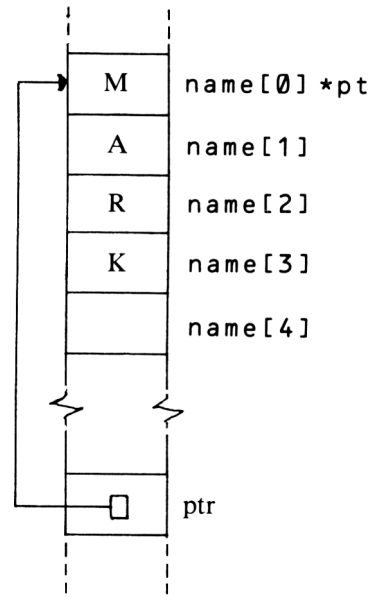
Pointers may be thought of as variables that contain the addresses of physical locations in the computer's memory. They are declared along with normal variables but are distinguished by preceding their name with a '*' character. The declared variable must have a data type that corresponds to the data that it points to. Whenever a pointer is used without the * it refers to the *contents* of the variable; i.e. the address of a physical location in the memory. When the * is included it refers to the *contents at the address* stored in the variable.

Example:

```
char name[20], *ptr;
ptr = &name[0];
*ptr = 'M';
```

is equivalent to

```
name[0] = 'M';
```



(Also remember that `&name[0]` could have been represented by `name`)

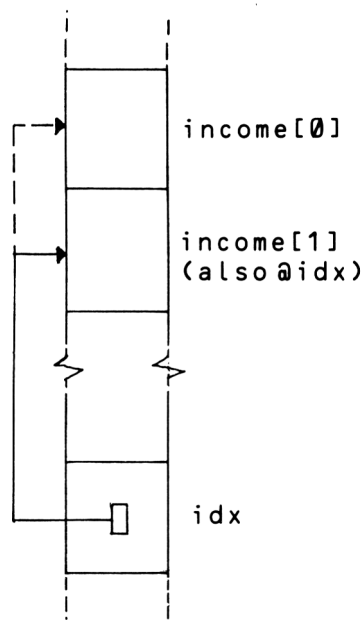
Example:

```
int s[10], *ptr;

ptr = s;          s;     ptr points to s[0].
(*ptr)++;                Increments s[0].
*(ptr*5) = 100;          Assigns 100 to s[5].
```

Mathematical operations on pointers are undertaken in multiples of the size of their data type. For example, if a long variable consists of four bytes, adding 1 to a long pointer increments the stored address by four bytes.

Example:

```
long income[12], *idx;
idx = income;
idx++;
```



`idx` now points to `income[1]`.

5.11 Pointer Arrays

Just as we have arrays of integers, characters, etc., we can also have arrays of pointers. This is illustrated in the following example which enables seven text strings to be accessed using an index.

Example:

```
char *day_text[] =
    {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
    };
```

The compiler assigns the address of the location where each string is stored to the corresponding element in the array of pointers enabling each day, 0 to 6, to be accessed via the appropriate pointer. We can even set up a pointer to point to the pointers!

As in the earlier examples, the array size is not specified so the complier counts up and uses the number of elements it detects.

5.12 Pointers to Functions

It is even possible in C to set up pointers to point to functions and they may be stored, manipulated and passed on to other functions in the same fashion as other pointers. However, we shall leave such ideas to later since the concepts can be rather mind boggling to the beginner.

5.13 Function Arguments

Earlier, we saw how programs are broken down into small functions each having a specific task, which may be used without worrying about the complexity of its operations. Once written, you need not know *how* it works – just that it does. C is designed so that the use of functions is easy and efficient enabling the programmer to keep control of potentially, very complicated problems.

Each function takes the following form:

```
function name (argument list)
    declaration of arguments
    {
    local declarations

    executable statements
    }
```

Arguments to functions provide a mechanism whereby functions can communicate with each other; the alternative method is to pass values via global variables. When a function is called, temporary variables, as declared before the opening left brace of the function, are set up and the passed arguments are assigned to each in turn. These variables are the function's own private copy for use during its lifetime, and are totally independent from any other variables.

A function may, if required, return a value or the result of an expression by using the `return` statement.

Program 4: Odd Numbers

```
main()
    {
    int x, y;

    y = 99;
    x = odd_test(y);
    }

odd_test(value)
    int value;
    {

    return (value & 1);
    }
```

Function `odd_test` returns a value of 1 if the argument is odd, or 0 if it is even. The method used is to mask the the least significant bit, which is set for odd numbers and unset for even numbers (see Chapter 4).

Functions returning useful values can be used as parameters to other functions.

Example:

```
process_data(ptr, odd_test(data) );
```

Functions returning data types other than `int` values require the data type to be specified at the start of the function. Those not returning a value may be defined as `void`.

Examples:

```
char toupper(c)
    char c;
    {
    ....
    ....
    }
```

```
void free(block)
    char *block;
    {
    ....
    ....
    }
```

Many applications require functions to affect the variables in the calling function; this can be done by passing pointers as parameters.

Program 5: pointer example

```
main()
    {
    int a, b;

    a = 10; b = 20;
    flip(&a,&b);
    }

flip(x,y)
    int *x,*y;
    {
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
    }
```

In this example, the addresses of the variables are passed and assigned to the pointer variables enabling the operations to be carried out on the locations where a and b are stored. This method can also be used for returning more than one value to calling functions.

Calls to functions which process arrays are controlled by passing the address of the first element of the array as a parameter. This is picked up by a pointer in the called function. The function can then access all elements in the array using offsets from this location. One common use of this is with string handling functions. So that functions can detect the end of the string, it is standard that all strings are terminated by a null character (character code of zero). String constants, (i.e a number of characters enclosed by double quotes), may be passed to functions in a similar fashion to passing the address of an array. The compiler passes the address of the internal representation of the constant and assumes that the string is terminated by a null.

Example:

```
char name[20];

strcpy(name,"STEPHANIE");
```

5.14 Variadic Functions

Hisoft C provides an addition to the syntax of function definitions and offers a method of defining functions which take a variable number of arguments (these are called *variadic* functions). The keyword `auto` can follow the function's argument list (before the function body) and causes the compiler to place the number of bytes of actual argument as an additional argument after the others. The function can access this argument and use it to work out how many arguments there are.

Example:

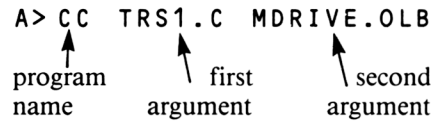
```
int max(param_byte_count) auto
/* from Hisoft library*/
    {
    static int argc, *argv, max;

    argc = param_byte_count / 2 - 1;
    argv = &param_byte_count + argc;     pointer to last
                                         argument.
```

5.15 Command Line Arguments

In some environments supporting C, there is method of passing arguments to the program when it is executed. These *command line arguments* are typed after

the program name that is to be executed.

Example:



Two arguments are passed to `main`; the first (by convention called `argc`) is the number of command line arguments, and the second (`argv`) is a pointer to an array of character strings.

Example:

```
main(argc, argv)
    int argc;
    char *argv[];
    {
    ...
    }
```

You are referred to the documentation supplied with your compiler to see if this feature is implemented. Command line arguments are not applicable to the Amstrad and ZX Spectrum versions of Hisoft C, and so will not be used in this book.

CHAPTER 6

MORE C STATEMENTS AND COMMANDS

6.1 Introduction

As we have already seen, a C program comprises a number of statements, each of which may do one of two things. It may either describe some item of information or it may tell the computer to undertake a specific operation.

The C statements may be divided up into five categories.

Those that we have met include:

1. DECLARATIONS
   Used to inform the compiler to reserve memory space for variables.

2. ASSIGNMENTS
   Used to allocate either numeric or character values to specific locations in the computer's memory.

3. FUNCTION CALLS
   Used to transfer control to other blocks of code; after execution control returns to the calling statement.

The remaining categories are:

4. PREPROCESSOR COMMANDS
   Control commands used to instruct the compiler of certain requirements during compilation.

5. CONTROL STATEMENTS
   Used to control the sequence in which the program commands are executed.

We shall now look at these last two in detail.
To clarify each C control statement, the syntax will be given in *italics.*

6.2 Preprocessor Commands

The preprocessor commands are not as clearly defined as the actual language since different systems have different requirements. The following commands are available with Hisoft C but may not be found on your version of C; likewise your version may contain others not discussed here, so you are advised to consult the documentation supplied with your compiler.

#define

The `#define` preprocessor allows *macro expansion.* The command is followed by an identifier text string and a constant. Whenever the identifier text string appears in the program, the compiler replaces it with the corresponding constant.

example:

```
#define BUFF_LEN 25

char msge_buffer[BUFF_LEN]; declares a character array of
                                                25 bytes
```

It is important to remember that everything after the identifier is transferred into the program (so do not include a semicolon unless you want it to reappear).

Some versions of C allow the macros to be passed arguments in a similar fashion to functions, but this is not available on the Hisoft implementation. Also not available with Hisoft C is the `#undef` preprocessor, which instructs the compiler to forget a previously defined macro.

The use of macros has two main advantages. Firstly it can help to improve the readability of a program and secondly, if any values have to be changed, they need only be amended at one place in the program.

#direct

The `#direct` preprocessor is followed by a "+" or a "–" and is used to toggle on/off *direct execution* mode. This facility is unique to Hisoft C and enables functions to be tested as they are written. When direct execution mode is enabled, a C command may be entered for immediate execution by typing just its name, any parameters and a semicolon.

example:

```
#direct #
printf( "Day no = %d ", calc_day_no("06/05/61")
);

(screen output)

#direct -
```

#translate

The `#translate` preprocessor enables object code that is produced from compilation to be saved to a specified filename such that it may be reloaded in *stand alone mode*, i.e. without the compiler, editor, etc. present. This is used for final versions of a program and is important for commercial distribution of your software products.

example:

```
#translate invaders          should be the first command in
                                                compilation
#include
```

#include

We met the `#include` preprocessor briefly in chapter 3 where we saw it was used to include a specified file into compilation.

Hisoft C has three major forms:

```
#include                     include source stored by the editor

#include filename            include specified source file
#include <filename>
#include "filename"

#include ?filename?          include functions in specified file that
                             have been previously invoked
```

#error

The `#error` preprocessor removes compiler error text messages from the RAM, once and for all, releasing about 2K of space.

#list

The `#list` preprocessor is followed by either "+" or "–" and is used to toggle on/off a compiler listing.

6.3 Control Statements

Probably some of the most vital statements in a computer's repertoire are those that allow us to control the order in which the instructions in a program are carried out. They enable the computer to become "intelligent" and able to test conditions so that, depending on results, different actions may be taken.

C contains a number of control statments which closely resemble the latter four types of PDL sub-units which we met in Chapter 3. They all involve testing logical conditions which, recapping on Chapter 4, evaluate to either zero or non-zero. C treats zero as a truth value False; non zero is considered as True. As you may remember, logical expressions can be made very complicated by linking conditions with the logical operators `&&`, `||` and `!`.

if ... then ... .

An `if` statement tests a condition within parenthesis and if True will execute the following statement (or, if within braces, the compound block). If required, an "else" statement may be included, in which case the following statement (or compound block) is executed if the condition is False.

if (condition)
statements

or

if (condition)
statements-1
else
statements-2

Example:

```
if (x == 0)
    process_0();
else
    if (x > 0)
        process_n(x);
    else
        error_msg("Invalid option");
```

switch ... case.

A frequent requirement is a multi-way decision whereby an expression is tested against a number of constant values. The `switch` statement provides a simple and efficient mechanism of coding such decisions. The result of the expression within parenthesis which follows the switch statement is used to select one of a number of `case` values. Execution continues from that point until a `break` statement is found, and then the switch is terminated. If none of the constant values matches, then a `default` label, if present, is selected. It is important

not to forget the `break` statement or the program will "plough" through the remaining selections.

```
switch (expression)
    {
    case <constant>:
         statements
    ......
    ......
    default:
        statements
    }
```

Example:

```
switch ( menu_option() )
    {
    case -1:
        help();
        break;
    case 0:
        return;
    case 1:
        option_1();
        break;
    case 2:
        option_2();
        break;
    case 3:
        option_3();
        break;
    default:
        error_msg("Invalid option");
    }
```

for

The `for` statement is used when we require a section of code to be repeated a specific number of times. The looping is controlled by making a once only number of initialisations. Then while a logical condition is True, the following statement or compound block is executed. On the completion of each loop, a number of expressions may be evaluated.

```
for (initialisation; condition; expressions)
    statements
```

The normal use is to assign a initial value to a control variable and then increment it on each loop until it exceeds a terminating value.

Example:

```
for (i = 0; i < 10; i++)
    process_option(i);
```

Occassionally, it is convenient to omit some of the components in the statement. In this extreme case, we have a "forever loop".

Example:

```
for ( ; ; )
    process_selection();
```

while

The `while` statement is similar to the for statement but only the condition is specified. The following statement or compound block is executed until the condition becomes False.

while (condition)
  statements

Example:

```
while (i--)                          execute function until i equals 0
    get_batch();
```

do

The while statement tests to see if the condition is True before any statements in the loop are executed. However, there are times when the loop is required to be executed at least once before any terminating test is made, and in such cases the `do` statement can be used. With this statement the `while` condition is positioned after the statement or compound block.

do
  statements
while (condition)

Example:

```
do
    get_data();
while (cont_flg)
```

continue and break

The statements `continue` and `break` may be used to take specific actions within loops.

Continue is used to skip to the end of a loop avoiding the following statements. Break causes immediate execution of the loop.

Another method of terminating a loop is the return statement which ceases execution of a procedure.

goto

Control may be transferred unconditionally, by use of the `goto` statement, to a defined label.

goto identifier

\- - - - - -

identifier:

Example:

NONE !! As discussed previously, the `goto` statement should be avoided. I have never used a C `goto` before and so I would rather not start now.

inline

Hisoft C provides a statement enabling the incorporation of machine code into a C program. This is useful for calling routines in the manufacturer's operating system that is stored in RAM.

inline (machine code values)

Examples:

```
#define CHAN_OPEN  0x1601
#define ld_a_with  0x3E
#define call       0xCD

inline (ld_a_with, 3, call, CHAN_OPEN);
```

call location 1601 (hex) with 3 assigned to register A

For more examples of the inline statement you are referred to the Hisoft library functions.

6.4. Some words of advice . . .

A beginner to C can read books on the subject until the print comes off the pages, but nothing will help him/her more than to seeing the language in action. Up to now, examples have been limited since we have not yet met any of the vital C library functions such as I/O routines. From now on our repertoire of C is sophisticated enough to illustrate everything with, hopefully, both interesting and useful programs.

Many of the functions given in this book will be used on several occasions but will only be listed once. It is suggested that as you progress through the book, you build up a library of useful procedures; you should include all those that are commented:

```
/* MRH.LIB */
```

as they will be required in later programs. Remember, they can be conditionally included for compilation by using:

```
#include ?MRH.LIB?
```

A complete list of functions in the MRH.LIB library is given in Appendix B.

CHAPTER 7

THE C STANDARD LIBRARY.

7.1 Introduction

We shall now look at the library functions which are provided with the Hisoft compiler. The majority of the functions we meet in this chapter are common to other versions of C. Most of the library procedures in this chapter are so fundamental to C programming that they tend to be thought of as part of the actual language.

The library is provided in three components; the header, the built-in functions and the remaining functions.

The header is a C source file 'stdio.h' containing constant and data type definitions and a couple of functions. Any program using the library must use this header by using the `#include` preprocessor at the start of the program.

i.e. `#include <stdio.h>`

The built-in functions are those frequently used and are contained, for efficiency, in the run-time system. They are automatically included in your compiled program.

The remaining functions are contained in a C source file 'stdio.lib'. They may be selectively compiled using the library search `#include` at the end of your program.

i.e. `#include ?stdio.lib?`

Other versions of C provide library functions in two files; a header that has to be included at the top of your source, and an object file containing the functions which have to be linked in at compilation time using facilities provided.

It is anticipated that Hisoft will supply additional library functions written by the users of their compiler at a low cost to make them widely available (provided free to those who supply functions for inclusion).

7.1.1 Function descriptions—an important note

For each procedure we shall meet, a few lines in *italics* are listed, which represent the top lines in the procedure's C code. From this it is clear what data type the procedure returns and what, if any, parameters are required (see section 5.13).

Example:

```
void error_message(msg)
    char *msg;
```

This procedure requires a character pointer to be passed and does not return any value, i.e. a type 'void'.

Example:

```
char *search(strg,num)
    char *strg;
    int num;
```

This procedure requires a character pointer and an integer as parameters (in that order) and returns a character pointer.

7.2 Simple I/O

Up to now, all of our examples have been hampered by the absence of any functions in our C repertoire for the output of data. Since it is pointless writing programs if we cannot access the results from the computer, we shall rectify our problem and study the most important output function, `printf`, which can be used for displaying all kinds of data, i.e. numbers, characters, strings, etc.

printf

The first parameter is the control string which, with the exception of *conversion specification* characters is displayed as it stands. Conversion specifications define the format for displaying the remaining arguments.

```
void printf(control, [ arg1, arg2, arg3, .... , argn ] )
    char *control;
```

Conversion specifications commence with a '%' character, followed by some *option modifiers*, and are positioned in the control string where the next successive argument is to appear.

The following conversion specifications are possible:

`%d` signed decimal notation
`%u` unsigned decimal notation
`%o` octal notation
`%x` hexadecimal notation
`%e` exponential notation
`%f` floating point notation

} not supported on Hisoft C

`%c` single character
`%s` string terminated by a null character
`%%` % character

example:

```
char  error[80];

printf("ERROR: %s", error);
```

example:

```
int  h_score, score;

printf("Your score is %d \n", score);
printf("Highest score is %d \n", h_score);
```

Some additional characters may appear between the `%` and the optional modifier; these are:

`-`	left justified (default is right)
`0`	leading zeros in field
`digit string`	minimum field width
`.digit string`	maximum number of characters printed from a string, or the number of decimal places printed in a floating point number
`L`	indicates long data type - not Hisoft C

example:

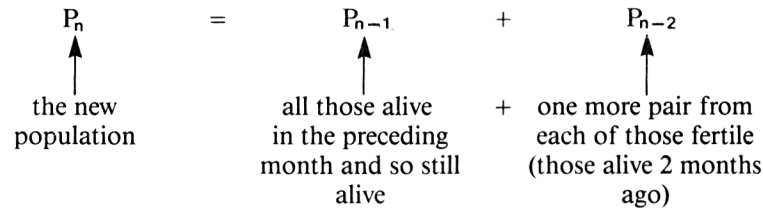
```
printf(":%-10.5s:","MICRODRIVE")
```

this would display `:MICRO      :`

Example: Fibonacci Sequence.

In the thirteenth century, the Italian mathematician Leonardo Fibonacci studied the vast explosion in the rabbit population. He considered how the population grew, starting with just one pair of rabbits and assumed that it took a pair of newborn rabbits one month before they became fertile and could reproduce. From then on they would breed an additional pair each month. He assumed that no deaths or migration occurred in the timespan under consideration. Analysing the problem it can be seen that if the population of

month n is P_n then:



The values of P_n are found in Program 6, the sequence of numbers obtained is known as the 'Fibonacci Sequence'.

Program 6:

```
/************************
*  FIBONACCI SEQUENCE   *
*----------------------*
*     M.R.H.    May 85   *
************************/

main()
    {
    int p,p1,p2,m;

    p1 = 1; p2 = 1;
    printf("\nFIBONACCI SEQUENCE");
    printf("\nMonth 1  Rabbits 1");
    printf("\nMonth 2  Rabbits 1");
    for (m = 3; m < 21; m++)
        {
        p = p1 + p2;
        printf("\nMonth %d  Rabbits %d",m,p);
        p2 = p1; p1 = p;
        }
    }
```

scanf

Equally important as accessing data is the input of information; the simplest method is via the `scanf` function which is the input analogue of `printf`. It provides many of the same conversion facilities but in the opposite direction.

int scanf(control, `[` *arg1, arg2, arg3, ... , argn* `]` *)*control;*
`gn] )`

All other arguments must be pointers!

`scanf` reads characters entered from the keyboard and interprets them as directed by the conversion specifications in the control string. The results are stored in the remaining arguments, all of which must be pointers (e.g. use the & operator on integer va iables). `scanf` stops when the control string is exhausted or when some characters are entered which do not match the control string. The function returns the number of input items that have been entered and which successfully match the control string; this value should be checked when the function returns and, if not as expected the function can be recalled.

In addition to the conversion specifications of `printf` (with the exception of `%u`), the control string may contain:

blanks	matched by any amount of *white space* (blanks, tabs or newlines)
ordinary characters	must match next input character
%* (suppression)	next input field is skipped
%digit	maximum field width

When `scanf` is called, the user may enter the inputs fields. The characters are *echoed* on the screen and may be amended using the cursor control and delete keys; the fields are accepted with the Enter key. The conversion specifications within the control field determine how each input field is interpreted. An input field is defined as a string of non white space characters or everything recieved until the field width, if specified, is exhausted. Thus Enter, Tab or Space may be used to separate fields for `scanf`.

example:

```
int  x; char  name[21];

printf("Enter data: ");
scanf("%d %20s",&x,name);
```

Program 7 responds to any entered month and year by displaying the calendar for that particular month. The original calendar was devised by Julius Caesar, but he fixed the length of a year to be eleven minutes too long. This error was rectified by the Gregorian calendar in Italy in 1582 although it was not introduced into England until 1752. The program will not give correct answers for earlier years.

The program works by using the function `calc_day_no` to calculate the day of the week (0-Sun, 1-Mon, ..., 6-Sat) for any date.

Program 7:

```
/***********************
*       CALENDAR        *
*-----------------------*
*    M.R.H.    May 85   *
***********************/

int m, y, d, dd, n;

main()
    {
    get_date();
    process_date();
    display_calendar();
    }

get_date()
    {
    printf("\nCALENDAR\n\n");
    m = 0; y = 0;
    do
        {
        printf("Enter month:");
        scanf(" %d",&m);
        }
    while (m < 1 || m > 12);
    do
        {
        printf("Enter year :");
        scanf(" %d",&y);
        }
    while (y == 0);
    }

process_date()
    {
    d = calc_day_no(1,m,y);
    if ( (m++) > 12 )
        {
        m = 1;
        y += 1;
        }
    dd = calc_day_no(1,m,y);
    if (dd < d)
        dd += 7;
    n = 28 + dd - d;
    }
calc_day_no(day,mon,yr)     /* MRH.LIB */
    int day,mon,yr;
    {
    int x1,x2,x3;

    mon = mon - 2;
    if (mon <= 0)
        {
        mon += 12;
```

```
        yr -= 1;
        }
    x1 = yr / 100;
    yr = yr - 100 * x1;
    x2 = (260 * mon - 19) / 100 + yr / 4 +
                              yr + x1 / 4 - 2 * x1 + day;
    x3 = (x2 < 0) ? x2 / 7 - 1 : x2 / 7 ;
    return (x2 - 7 * x3);
    }

display_calendar()
    {
    int j, p;

    printf("\n SUN  MON  TUE  WED  THU  FRI  SAT\n\n");
    for (j = 0; j < d; j++)
        printf("     ");
    p = d;
    for (j = 1; j <= n; j++)
        {
        printf("  %2d ",j);
        if (p == 6)
            {
            printf("\n");
            p = 0;
            }
        else
            p++;
        }
    }
```

7.3 Character tests

There are a number of standard C functions for determining the nature of a character.

isalnum

Returns True if the character is alphanumeric (i.e. a letter or a digit), otherwise it returns False.

int isalnum(c)
char c;

Example:

```
char  *ptr;

if (isalnum(*ptr))
    menu_one();
else
    print f("Invalid option \n");
```

isalpha

Returns True if the character is a letter otherwise, it returns False.

```
int isalpha(c)
    char c;
```

isascii

Returns True if the character code is less than 128, otherwise it returns False.

```
int isascii(c)
    char c;
```

iscntrl

Returns True if the character is a control character, otherwise it returns False.

```
int iscntrl(c)
    char c;
```

isdigit

Returns True if the character is a digit, otherwise it returns False.

```
int isdigit(c)
    char c;
```

islower

Returns True if the character is lower case, otherwise it returns False.

```
int islower(c)
    char c;
```

isprint

Returns True if the character is printable, otherwise it returns False.

```
int isprint(c)
    char c;
```

ispunct

Returns True if the character is printable but not a letter or digit, otherwise it

returns False.

int ispunct(c)
char c;

isspace

Returns True if the character is a white space character, otherwise it returns False.

int isspace(c)
char c;

isupper

Returns True if the character is an upper case character, otherwise it returns False.

int isupper(c)
char c;

7.4 Character and String Manipulation

The C language treats all strings as arrays of characters and so a number of fundamental library functions exist to deal with them.

toupper

If the passed character is in lower case then the character is returned in upper case; other characters are returned unchanged.

char toupper(c)
char c;

Example:

```
for (j = 0; name[j] != '\000'; j++)
    name[j] = toupper(name[j]);
```

tolower

If the passed character is in upper case then the character is returned in lower case; other characters are returned unchanged.

char tolower(c)
char c;

strlen

Returns the length of a string, i.e. the number of bytes between the passed address and the first null character.

unsigned strlen(s)
*char *s;*

example:

```
char buffer[80];
int len;

len = strlen(buffer);
```

strcpy

Copies one string to another, i.e. copies from the passed source address to another address, byte by byte, until the first null character is detected.

*char *strcpy(dest,source)*
*char *dest, *source;*

Example:

```
char car[6];

strcpy(car,"VOLVO");
```

strcat

Concatenates one string onto the end of another, i.e. copies from a passed address, byte by byte to the end of a string starting at a base address. Returns a pointer to the start of the base string.

*char *strcat(base,add)*
*char *base, *add;*

example:

```
char error_msg[80], project[15];

strcpy(error_msg,"Project ");
strcat(error_msg,project);
strcat(error_msg," does not exist");
```

strcmp

Compares two strings, byte by byte, and returns a value:

0 if the strings are identical
> 0 if the first string is greater than the second string
< 0 if the first string is less than the second string

A string is found to be less than another if the first character that differs comes earlier in the ASCII character set.

char strcmp(s,t)
*char *s, *t;*

example:

```
char  response[20];

if ( strcmp("END",response) == 0)
    return;
```

Other String Functions.

Absent from early versions of Hisoft C are the following standard C functions:

*char *strncpy(dest,source,n)* *char *dest, *source;* *int n;*	string copy up to a maximum of n characters
*char *strncat(base,add,n)* *char *base, *add;* *int n;*	string concatenate up to a maximum of n characters
*char *strncmp(s,t,n)* *char *s, *t;* *int n;*	string compare up to a maximum of n characters

However, any Hisoft user that requires these functions should not have any problems in writing them and can include them in their own C function library.

Program 8 has been devised for a hypothetical software house which has a large number of staff, each with various skills and experience of different machines, languages and operating systems. The staff scheduler needs a means of keeping a record of each employee's skills and a method of obtaining the names of all those who are familiar with a specified subject. Thus, some sort of database is required.

The names of the employees are stored as strings followed by their computer skills. To distinguish between a name and a skill, the name is preceded by the symbol '#'. The program does a word search on each item; if a word is preceded by '#' then the employee's name is temporarily stored while his/her skills are examined. If a required skill is found, then the employee's name can be displayed. The program then continues to search through the remaining items until the termination symbol '%' is detected.

This is a very elementary form of database. The staff scheduler can easily add to his records by amending the initialisations in the program. You could change these statements to set up a database with something more relevant to your own needs, for example, a book or record library.

Program 8:

```
/***********************
*      DATA SEARCH       *
*-----------------------*
*     M.R.H.    May 85    *
***********************/

#include <stdio.h>

#define TRUE        1

char *data[]=
   {
   "#BROWN N","ADA","FORTRAN","VAX",
   "#CLOUGH S","BASIC","COBOL","DEC","TAL",
   "#FIELDS D","CORAL","COBOL","ARGUS","ASSEMBLER",
   "#HARRISON M","C","CP/M","UNIX","TANDEM","COMMS",
   "#MACALISTER C","BASIC","COBOL","PDP","ARMY",
   "#SINCLAIR M","ALGOL","COBOL","BURROUGHS",
   "#STEVENS G","CAD/CAM","UNIX","PROLOG","MICROCOBOL",
   "#THOMPSON P","PASCAL","POLICE",
   "#YATES D","BASIC","Z80","68000","CP/M",
   "%",
   };

main()
   {
   char skill[55];

   printf("SOFTWARE SKILLS LTD");
   while (TRUE)
      {
      printf("\n\nEnter skill required: ");
      scanf(" %s",skill);
      if (strcmp(skill,"bye") == 0)
         return;
      process_data(skill);
      }
   }
process_data(ptr)
    char *ptr;
    {
```

```
    int j;
    char name[25];

    for (j = 0; *(ptr +j) != '\000'; j++)
        *(ptr + j) = toupper(*(ptr +j));
    for (j = 0;  ; j++)
        {
        switch (*data[j])
            {
            case '#':
                strcpy(name,data[j] + 1);
                break;

            case '%':
                return;

            default:
                if (strcmp(data[j],ptr) == 0)
                    printf("\n%s",name);
                break;
            }
        }
    }

#include ?stdio.lib?
```

7.5 Sorting Data

One of the few advantages a computer has over the human brain is its ability to undertake long and laborious, but relatively simple tasks, in a short time. One such task we shall examine is the mundane job of sorting data comprised of numeric or alphabetic items into some sequence – usually increasing or alphabetical order. There have been many algorithms devised for doing such operations and their efficiency is usually proportional to their complexity.

Most C libraries contain at least one sorting function – the function provided by Hisoft is called `qsort` and uses the well known *shell sort* algorithm.

The simplest method of sorting is called a *bubble sort* because the lowest values can be though of as floating upwards to one end of an array and the highest values sink to the bottom. The array is continually scanned and two neighbouring items are swapped if the first has a higher value than the other; if no exchanges are made, the list is in order.

Whereas the bubble sort compares adjacent data items in the array, a shell sort makes initial comparisons between items that are far apart, on the assumption that if elements are far apart and have to be swapped it is more efficient to do it as soon as possible. The separation between data elements is called the *sort interval*. Initially, the sort interval is set to the number of elements and then on

each scan the interval is reduced by one half until the final scan is equivalent to a bubble sort.

void qsort(list,no_items,size,comp_function)
*char *list;*
int no_items, size;
*int (*comp_funtion)();*

Way back in Chapter 5, it was said that we could set up pointers to functions and manipulate them like any other pointer. This is now illustrated since the fourth parameter to `qsort` is a pointer to a function which compares two items in the list, and returns values that are positive, zero or negative according to whether the first item is greater than, equal or less than the second. If the sort is alphabetical on strings, then the `strcmp` function can be used.

The most common structure for the list is a two dimensional array which is `no_items` long and `size` bytes wide.

```
char  list[no_items][size];
```

All of this is now demonstrated in Program 9 which allows a number of strings to be entered. To terminate string input and sort the list, type `[CTRL] Z` followed by `[ENTER]`; the sorted list will then be displayed.

Program 9:

```
/***********************
*        SORT           *
*-----------------------*
*    M.R.H.    May 85   *
***********************/

#include <stdio.h>

#define LINEFEED      '\012'
#define MAX_NO_ITEMS  100
#define SIZE          25
#define stdin         0

char list[MAX_NO_ITEMS][SIZE];
int count;
extern int strcmp();

main()
   {
   get_strings();
   printf("\nSorting\n");
   qsort(list,count,SIZE,strcmp);
   display_strings();
   }

get_strings()
   {
```

```
    for (count = 0; count < MAX_NO_ITEMS; count++)
        {
        printf("%d: ",count);
        fgets(list[count],SIZE,0);
        if (list[count][0] == LINEFEED)
            break;
        }
    }

display_strings()
    {
    int j;

    for (j = 0; j < count; j++)
        printf("%s",list[j]);
    }

#include ?stdio.lib?
```

7.6 Arithmetic Functions

There are several arithmetic functions available to the C programmer:

max

Returns the greatest value in the set of arguments. This function is variadic (i.e. can have any number of arguments).

int max(arg1, arg2, ... , argn) auto

min

Returns the smallest value in the set of arguments. This function is variadic.

int min(arg1, arg2, ... , argn) auto

abs

Returns the absolute value of its argument.

int abs(n)

sign

Returns one of the following values:

1 if the argument is positive

0 if the argument is zero
−1 if the argument is negative

int sign(n)

Recursion

The following program shows a technique called *recursion* in which a function calls itself. This essentially offers no new ideas, but can often lead to neat and elegant solutions. The example uses a famous algorithm, named after a mathematician called Euclid, for finding the highest common factor of two integers. Remember, unsigned variables are restricted to the range 0 to 65535.

Program 10: highest common factor

```
/***********************
*       EUCLID          *
*-----------------------*
*    M.R.H.    May 85   *
***********************/


main()
   {
   unsigned x,y;

   printf("Enter X and Y:");
   scanf("%d %d",&x,&y);
   printf("Highest common factor = %d", hcf(x,y) );
   }

hcf(m,n)
   unsigned m,n;
   {
   if (m < n)
      hcf(n,m);
   else
      if (n == 0)
         return m;
      else
         hcf(n,m % n);
   }
```

7.7 Format Conversion Functions

A number of routines are provided in the C libraries for converting numbers in ASCII character format to an appropriate data type.

atoi

Scans the string starting at a passed address and, after ignoring any initial white

space, it converts the characters, up to the first non digit, into their equivalent integer form. Numbers may have a leading '+' or '–'. A value of 0 is returned if no number is present.

int atoi(s)
*char *s;*

example:

```
char  menu_select[3];
int  selection;

scanf("%2s",menu_select);
selection = atoi(menu_select);
```

Other format conversion functions

Since floating point and long arithmetic are not yet supported by Hisoft, the following standard C functions are absent from their implementation.

double atof(s)
*char *s;* — converts string s to a double precision floating point number

long atol(s)
*char *s;* — converts string s to a long number

7.8 32 Bit Number Arithmetic.

Long Functions.

Relevant to Hisoft users only, are a number of functions providing facilities for long arithmetic. The numbers are represented by pointers to arrays of four bytes, in which the least significant byte (l.s.b.) is stored in element 0, and the most significant byte (m.s.b.) is stored in element 3. All arithmetic is unsigned.

The functions are:

void long__multiply(c,a,b)
*char *c,*a,*b;* — multiply c = a * b

void long__add(c,a,b)
*char *c,*a,*b;* — add c = a + b

void long__init(a,n1,n2)
*char *a;*
int n1, n2; — initialise upper 16 bits with n1 and lower 16 bits with n2

void long__set(a,n,d)
*char *a;*
int n,d; — initialise 16 bits at bytes d and d+1

void long__copy(c,a)
*char *a,*c;* — assign c = a

Pseudo-Random Number Generator

These long functions have been used by the Hisoft random number generator which is extremely useful for games requiring an element of luck. It should be realised that the number generator is not truly random since it works by returning sequentially values from a very long list of numbers. However, since the list is so long, a random number of sorts can be achieved by starting at an unknown position and this is usually more than adequate for most requirements. The position in the sequence is dependent upon a value called the *seed*. If the seed is reset to the same value each time the random number generator function is called, then the same random number will be returned. (Most other compilers have some equivalent method of generating random number)

The functions are:

void srand(n)
int n; — 'seeds' the generator with a value n

int rand() — returns a 16 bit pseudo random number

Program 11 demonstrates how setting the seed to the same value produces the same sequence of numbers. The program is used to code and decode secret messages, and works in a similar method to the German "Enigma" coding device used in World War Two. That machine was dependent upon an entered "codeword", and would then code messages by replacing the characters in the message by other characters. To make things more difficult, a character appearing more than once would usually be replaced by different characters on each occurrence.

The program uses the random number generator to choose replacement characters. The basic idea is that when the generator is re-seeded with the same

value, it starts at the same position in the sequence list. The seed used is always dependent upon the characters in the codeword so, when sending the message, it is important that the receiver knows the codeword so that he can set his random number generator to the correct starting position. All spaces in the message should be represented by the underscore character "_".

Program 11:

```
/**********************
*       ENIGMA          *
*----------------------*
*    M.R.H.    May 85   *
**********************/

#include <stdio.h>

#define TRUE            1
#define FALSE           0
#define BELL          '\007'
#define MESSAGE_LEN     240

int len, j;
char codeword[20], message[MESSAGE_LEN];
char chrctrs[]  = "_0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

main()
   {
   printf("ENIGMA");
   while (process())
      ;
   }

process()
   {
   char opt;

   while (TRUE)
      {
      printf("\n\nEnter: (C)ode, (D)ecode or e(X)it - ");
      scanf(" %c",&opt);
      switch(opt)
         {
         case 'c':
         case 'C':
            code_msg();
            return TRUE;
         case 'd':
         case 'D':
            decode_msg();
            return TRUE;
         case 'x':
         case 'X':
            return FALSE;
         default:
            printf("Invalid option");
            bell();
         }
      }
   }
```

```
code_msg()
   {
   int p;

   reset_codeword();
   input_msg();
   printf("Coded message: ");
   for (j = 0; j < len; j++)
      {
      p = position(message[j]) + rnd(37);
      if (p > 36)
         p -= 37;
      printf("%c",chrctrs[p]);
      }
   }

decode_msg()
   {
   int p;

   reset_codeword();
   input_msg();
   for (j = 0; j < len; j++)
      {
      p = position(message[j]) - rnd(37);
      if (p < 0)
         p += 37;
      printf("%c",chrctrs[p]);
      }
   }

position(c)
   char c;
   {
   if (c < 48 || c > 90)
      c = 47;
   if (c > 64 && c < 91)
      c = c - 7;
   return (c - 47);
   }

input_msg()
   {
   int cont_flg;

   cont_flg = TRUE;
   while (cont_flg)
      {
      cont_flg = FALSE;
      printf("Enter message: ");
      scanf(" %s",message);
      len = strlen(message);
      for (j = 0; j <len; j++)
         {
         if (isalnum(message[j]) || message[j] == '_')
            message[j] = toupper(message[j]);
         else
            {
            printf("Invalid character %c\n",message[j]);
            cont_flg = TRUE;
            }
         }
      }
```

```
    }

reset_codeword()
    {
    for (j = 0; j < 4; j++)
       codeword[j] = NULL;
    printf("\nEnter codeword: ");
    scanf(" %s",codeword);
    srand(8 * codeword[2] + 4 * codeword[1] + codeword[0]);
    }

bell()     /* MRH.LIB */
    {
    printf("%c",BELL);
    }

rnd(arg)   /* MRH.LIB */
    int arg;
    {
    unsigned x, y;

    x = rand() & 255;
    y = ( arg * x ) / 256 + 1;
    return y;
    }

#include ?stdio.lib?
```

7.9 Memory Management

When a C program is executed, some of the memory is used for the program code, some for the permanent data and some is reserved for the *stack* which stores local data and so expands and shrinks as functions are called and exited. There is often additional memory which can be accessed via memory management functions which reserve and release the space.

calloc

Allocates memory space for a specified number of items of a given byte size. It either returns a pointer to the start of the section of memory or otherwise NULL (value 0 as defined in the standard header) if there is no space available. A number of hidden control bytes are stored in front of the allocated block and must not be changed; thus you should remember overheads are involved. The allocated block is sometimes called a *heap*.

*char *calloc(n,size)*
unsigned n,size;

example:

```
char *ptr;
```

```
ptr = calloc(250,1);
```
reserves 250 bytes

free

Returns an allocated heap to the free store enabling future re-allocation. The pointer returned from calling `calloc` must be passed and the control bytes at the start of the heap must not have been corrupted.

sbrk

Before any heaps can be allocated, a section of the memory has to be reserved. On larger systems, such as those using Unix, this function would call the operating system, which would then make the memory available by moving other items about in the memory. On smaller microcomputers it is up to the user to decide on a safe place, away from the program and important data. By consulting your microcomputer's documentation you may find that there is a way of reserving such space before you start up your compiler. The alternative, and probably the safest method is to define a large static variable and use the space that it creates for heap management.

The Hisoft implementation, which uses the latter of these two methods, requires the number of bytes of physical memory that can be used by `calloc` to be passed as an argument.

```
char *sbrk(n)
    unsigned n;
```

The maximum size of the heap area is defined by a `#define` statement in the library and so may be amended if required.

swap

Swaps the contents of two areas of memory each a specified byte length long.

```
void swap(ptr1,ptr2,length)
    char *ptr1,*ptr2;
    unsigned length;
```

`swap` is used by the sort function `qsort`.

blt

Moves the contents of one area of memory of a specified byte length to another area. The move is undertaken in a non-destructive direction – which is important if the areas overlap. Whilst this function is similar to `strcpy`, it

always copies the given number of bytes whereas the string version terminates at the first null character. Also `blt` is more efficient since it is built-in.

peek

Returns the byte value at a specified address.

```
char peek(address)
    unsigned address
```

poke

Puts the least significant byte of a passed value at a specified address.

```
void poke(address,value)
    unsigned address, value;
```

Whilst the functions peek and poke aid the readability of a program, it is far more efficient to use a character pointer which can point to an address, and then read and assign values directly.

7.10 Advanced I/O

The C language has no facilities for input and output, and so accessing the display screen, the keyboard and cassette or floppy disk files is done by means of standard I/O functions. The I/O functions that we shall meet exist in a compatible form with any C implementation, and so, restricting all your I/O to these routines enables your software to be easily ported onto other systems. Besides, the functions provided are so versatile, it is unlikely that anyone would need to write their own.

All I/O in C is done via files, with devices treated as special cases. A file is a sequence of bytes which can be read or written to one at a time (known as *serial access*). A file may exist on cassette, floppy disk, microdrive or may be a device such as the screen or a printer. There are three standard files called `stdin` (*standard input*), `stdout` (*standard output*) and `stderr` (*error output*). Usually, the standard input file is the keyboard and the standard and error output files are the screen.

fopen

Before a file may be used it has to *opened* using the `fopen` function; this specifies a file name and a mode. The mode is a string specifying the action

which is to be carried out on the opened file:

"r" reading
"w" writing
"a" appending

If a file is successfully opened, the function returns a pointer to a block of data, found in the header, which contains essential information about the file. A new data type called FILE, which is also defined in the header, is used for file pointers. If any errors occur when opening the file, a value NULL is returned.

Due to the nature of the disk operating systems on the Amstrad and ZX Spectrum, the Hisoft `fopen` modes are restricted to "r" and "w", and opening an existing file for writing will first cause the file to be erased. Also, you are limited to just one input and one output file open at any one time. By inspecting the standard header, you will also see that the FILE definition is just a single integer value, but in order to keep our software portable, we shall use the conventional FILE notation.

*FILE *fopen(name,mode)*
*char *name,*mode;*

example:

```
FILE  *fp;

if  ((fp = fopen("STAFF.DAT","r"))  ==  NULL)
    {
    printf("Failure opening STAFF.DAT");
    return -1;
    }
```

fclose

When all operations with a file have been completed, the file must be closed using `fclose`.

int fclose(fp)
*FILE *fp;*

When data is loaded from a file, one block is loaded at a time and stored into a buffer, and then the data is accessed from the buffer. Similarly, when data is being stored in a file it is stored in the buffer, and sent to the cassette unit or disk drive when the buffer becomes full. Further data is sent by refilling the buffer. The use of the buffer avoids the drive or cassette unit being continually started

and stopped for each data item, and so speeds up processing and enables data to be more closely packed. When writing to files it is vital to close the file, since this function transmits the final data by flushing the output buffer.

getc

A single byte value may be read from a file using `getc`.

int getc(fp)
*FILE *fp;*

When the end of file is reached, the function returns a value EOF (−1, as defined in the standard header). When testing for EOF, it is important to recognise that the function is returning an integer value. You must not first assign the returned value to a character since the most significant byte will be lost, thus making the comparison always False.

Program 12 demonstrates the previous I/O functions with a useful program for dumping the contents of a file in both ASCII and hexadecimal form. Such a program is often very useful for examining the contents of files when debugging your software.

Program 12: file dumper

```
/***********************
*       FILE DUMP       *
*-----------------------*
*    M.R.H.    May 85   *
***********************/

#include <stdio.h>

#define TRUE            1
#define FALSE           0

char filename[13], data[16];
FILE *fp;
int c, idx, cnt;

main()
   {
   get_file();
   idx = 0;
   cnt = 0;
   while ((c = getc(fp)) != EOF)
      {
      data[idx] = c;
      idx++;
      if (idx == 16)
         print_line();
      }
   if (idx != 0)
      print_line();
   }
```

```
get_file()
    {
    printf("Enter file: ");
    scanf(" %12s",filename);
    if ((fp = fopen(filename,"r")) == NULL)
        {
        printf("\nFailure to open %s",filename);
        return;
        }
    }

print_line()
    {
    int j;

    printf("\n%04x: ",cnt);
    cnt += 16;
    for (j = 0; j < 16; j++)
        printf(" %02x",data[j]);
    printf("     ");
    for (j = 0; j < 16; j++)
        {
        if (isprint(data[j]))
            printf("%c",data[j]);
        else
            printf("?",data[j]);
        }
    for (j = 0; j < 16; j++)
        data[j] = 0;
    idx = 0;
    }

#include ?stdio.lib?
```

ungetc .

The **ungetc** function is used to put back a character onto a file so that it becomes the next character to be read with getc() or getchar().

int ungetc(c,fp)
int c;
*FILE *fp;*

putc

A single byte may be written to a file using `putc`.

int putc(c,fp)
int c;
*FILE *fp;i*

getchar

Since input from the keyboard is so common, the function `getchar` has been

provided for convenience to input one character. This function does buffered input, that is the characters are collected into an input buffer until the Enter key is pressed. If required, the Delete key may be used to edit the characters as they are typed. All characters entered are echoed on the screen.

int getchar()

example:

```
printf("Hit ENTER to continue");
getchar();
```

putchar

Again for convenience, a function `putchar` is provided to display characters on the screen.

int putchar(c)
int c;

example:

```
bell()
    {
    putchar(7);
    }
```

fgets

The `fgets` function reads a string of a maximum length from a specified file.

*char *fgets(s,n,fp)*
*char *s;*
int n;
*FILE *fp;*

The input stops when either n-1 characters have been read, or when a newline character is found. The input string is terminated with a null character. The function normally returns the address of the input string, but if the end of file has been reached when the function is called, the value NULL is returned.

Example:

```
fgets(buffer,51,fp);
```

fputs

The `fputs` function writes a string to a specified file.

```
void fputs(s,fp)
    char *s;
    FILE *fp;i
```

gets

The Hisoft implementation provides a function `gets` to read from the standard input, i.e. the keyboard. It is similar to fgets except that no maximum length is specified and the newline character is overwritten by a null character.

```
char *gets(s)
    char *s;
```

puts

Also provided by Hisoft, is a function `puts` to output a string to the standard output, i.e. the screen.

```
void puts(s)
    char *s;
```

fprintf

Acts like the function `printf` except output is directed to a specified file.

```
void fprintf(fp,control,arg1, ... , argn)
    FILE *fp;
    char *control;
```

sprintf

Acts like the function `printf` except output is directed to a specified string.

```
void sprintf(s,control,arg1, ... , argn)
    char *s, *control;
```

fscanf

Acts like the function `scanf` except input is taken from a specified file.

```
int fscanf(fp,control,arg1, ... , argn)
    FILE *fp;
    char *control;
```

sscanf

Acts like the function `scanf` except input is taken from a specified string.

```
int fscanf(s,control,arg1, ... , argn)
    char *s,*control;
```

exit

Terminates execution and closes all files which the program has opened. With the Hisoft implementation, the value passed causes the corresponding Amstrad or ZX Spectrum error report to be given.

```
void exit(n)
    int n
```

File Useage

In most computer systems, data may be organised either in *sequential* or *random access* files. When sequential files are used, the file has to be read through from the beginning until the required record is found; in random access files any record can be accessed directly, with almost equal speed. However, due to limitations with the Amstrad and ZX Spectrum systems, all data files must be organised sequentially. To use sequential files we must be able to:

1. Set up a file.
2. List the file.
3. Access records in the file.
4. Delete records in the file.
5. Amend records in the file.
6. Add records to the file.

These six operations are illustrated in program 13 which records names and telephone numbers. The data is first loaded into an array and then searches and amendments can be made. Finally, when all operations are complete, the file on the disk may be updated.

Program 13: telephone index

```
/***********************
*  TELEPHONE NUMBERS    *
*----------------------*
*     M.R.H.    May 85   *
***********************/

#include <stdio.h>

#define TRUE            1
#define FALSE           0
#define stdin           0
#define MAX_NO_ITEMS    100

char data[MAX_NO_ITEMS][42], *ptr;
char name[21], phone[21];
FILE *fp;
int j, count;

main()
   {
   if (read_records())
      {
      main_process();
      write_records();
      }
   else
      create_file();
   }

main_process()
   {
   char opt[3];

   while (TRUE)
      {
      printf("\n\nTELEPHONE NUMBERS");
      printf("\n\n1. List records");
      printf("\n2. Access record");
      printf("\n3. Add record");
      printf("\n4. Delete record");
      printf("\n5. Exit");
      printf("\n\nEnter facility required: ");
      fgets(opt,3,stdin);
      switch(opt[0])
         {
         case '1':
            list_records();
            break;
         case '2':
            access_record();
            break;
         case '3':
            add_record();
            break;
         case '4':
            delete_record();
            break;
         case '5':
            return;
         default:
```

```
            printf("Invalid option");
            bell();
          }
       }
    }

read_records()
    {
    int cont_flg;

    ptr = data[0];
    printf("\nReading file:");
    if ((fp = fopen("PHONE.DAT","r")) == NULL)
       {
       printf("\nFailure to open PHONE.DAT");
       return FALSE;
       }
    fscanf(fp,"%d ",&count);
    printf(" %d records read",count);
    for (j = 0; j < 42 * count; j++)
      *(ptr + j) = getc(fp);
    fclose(fp);
    return TRUE;
    }

create_file()
    {
    char opt;

    printf("\nDo you want to create PHONE.DAT (y or n)");
    scanf(" %c",&opt);
    if (opt == 'y' || opt == 'Y')
       {
       fp = fopen("PHONE.DAT","w");
       fprintf(fp,"%d ",0);
       fclose(fp);
       }
    }

write_records()
    {
    printf("\nWriting file:");
    fp = fopen("PHONE.DAT","w");
    fprintf(fp,"%d ",count);
    printf(" %d records written",count);
    for (j = 0; j < 42 * count; j++)
       putc(*(ptr+j),fp);
    fclose(fp);
    }

list_records()
    {
    printf("\nPhone List\n");
    for (j = 0; j < count; j++)
       {
       if (data[j][0] != '\377')
          printf("\n%d: %s   %s",j,&data[j][0],&data[j][21]);
       }
    }

access_record()
    {
    printf("\nAccess record\n\n");
```

```
printf("Enter name:  ");
fgets(name,21,stdin);
for (j = 0; j < count; j++)
    {
    if (strcmp(data[j],name) == 0)
        {
        printf("Found phone: %s",&data[j][21]);
        return;
        }
    }
printf("Not found\n");
}

add_record()
    {
    printf("\nAdd record\n\n");
    printf("Enter name:  ");
    fgets(name,21,stdin);
    printf("Enter phone: ");
    fgets(phone,21,stdin);
    for (j = 0; j < count; j++)
        {
        if (data[j][0] == '\377')
            {
            strcpy(&data[j][0],name);
            strcpy(&data[j][21],phone);
            return;
            }
        }
    strcpy(&data[count][0],name);
    strcpy(&data[count][21],phone);
    count++;
    }

delete_record()
    {
    printf("\nDelete record\n\n");
    printf("Enter name:  ");
    fgets(name,21,stdin);
    for (j = 0; j < count; j++)
        {
        if (strcmp(data[j],name) == 0)
            {
            data[j][0] = '\377';
            printf("Deleted\n");
            return;
            }
        }
    printf("Not found\n");
    }

#include ?mrh.lib?

#include ?stdio.lib?
```

CHAPTER 8
DATA STRUCTURES

8.1 Introduction

In this chapter we shall see how to organize complex data so that we can handle it in a convenient fashion. This involves collections of variables, often of different data types, grouped together under a single name. As we shall see, C provides powerful mechanisms for defining groups of data and manipulating them through the use of pointers.

Before we move on to such topics, we shall first take another look at C data types and see how we can define new ones (other than `int`, `char`, etc).

8.2 More on C Data Types.

Earlier in Chapter 5, we saw that there are six basic data types in C:

`char, int, short, long, float, double.`

All variables must be declared before use. (A declaration specifies the data type followed by a list of variables of that particular type.)

When operands of different types appear in the same expression, a few simple C rules ensure that the data is converted to a common type. Usually the conversion only takes place if the expression makes sense; for example, using a `float` type for an array subscript would not be allowed.

`chars` and `ints` may be used together freely within the same algebraic expressions, since the `char` values are automatically converted to `int` type. With most compilers, `chars` with values above 127 are converted as positive numbers, hovever, care should be taken as there are a few versions which convert such values as negative numbers.

In general, if two operands of different data types are involved in an arithmetic

operation, the following sequence of operations takes place.

`char` and `short` are converted to `long`;
`float` is converted to `double`;
If either operand is `double`, the other is converted to `double` and the result returned is `double`;
Otherwise, if either operand is `long`, the other is converted to `long` and the result returned is `long`;
Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and the result returned is `unsigned`;
Otherwise, the result returned is `int`.

Conversion takes place across assignments and when passing arguments to functions. For example, if a `char` value is assigned to an `int` variable, the value is converted to an `int` first and then stored in the variable. An `int` value can be assigned to a `char` variable but the most significant byte is discarded. Likewise, converting a `long` value to an `int` is done by discarding the most significant bits. Converting `float` to an `int` causes the fractional component to be truncated and lost, and `double` to `float` conversion is done by rounding.

It is possible to force or *coerce* any expression into a specific data type by using a construction called a *cast*. Casting a value to another data type is done by including the data type within brackets before the value.

example:

```
int x;
sqrt( (double) x);
```

New data type names can be created using the `typedef` facility. For example, if we wanted a new data type `COUNT` which would store integer values we would declare:

```
typedef  int  COUNT;
```

From then onwards, the type `COUNT` could be used in declarations and casts in the same fashion as `int` would be used.

Another commonly used example is `STRING`:

```
typedef  char *STRING;
```

makes `STRING` a synonym for `char *` and so can be used in declarations requiring string pointers.

The use of `typedef` declarations has two advantages, firstly it can aid porting software onto new environments where certain data types may be machine specific, and secondly it can aid the readability of a program.

In order to simplify the Hisoft compiler (and make programs more readable) casts must be preceded by the keyword `cast` and use types that are predefined or defined by a `typedef` statement.

i.e. *cast(type_specifier) expression*

With C it is possible to store different kinds of data in the same section of memory and let the compiler handle the size and alignment problems. This is done by using *unions* which are variables that can store data of different types but, of course, not at the same time. It is up to the logic in the program to know what type of data is currently stored. A union has to be declared just like any other variable. This involves listing the different types that the union can store (*members*) and declaring the overall variable name. The compiler allocates enough space for the largest data item required.

example:

```
union
    {
    int  ival;
    char  cval;
    long  lval;
    } total;
```

When used, the individual members of the union are accessed by using both the union and member names.

example:

```
total.ival = 100;
total.lval = 100L;
```

The only operations permitted on unions is accessing a member and obtaining its address. More complicated operations, such as passing to functions, has to be undertaken using pointers.

8.3 Structures

It is a common requirement in programming to group several items of related data together. For example, when considering a staff list, the information may include name, personnel number, grade, salary, holiday taken/entitlement, etc.,

all of which are best considered as one unit rather than separate entities. The C language makes this simple enabling *structures* to be defined which contain all of the data items required.

The first stage in using structures is to define a template which lists each of the data items in the structure. Creating a template does not occupy any physical space in the memory, but allows us to later declare any number of structures of that format.

example:

```
struct staff_tmp
    {
    char  st_name[20];
    char  st_pers_no[6];
    char  st_grade[2];
    int   st_salary;
    int   st_hol_tak;    /* half days */
    int   st_hol_ent;
    char  st_comment[30];
    };
```

The name of the structure template, in this case `staff_tmp`, is known as the *structure tag*.

Once the template has been defined, structures may be declared just like any other variable. With structure declarations the keyword `struct` is followed by the structure tag and then the list of structure names.

example:

```
struct  staff_tmp  write_staff_rec,
                   read_staff_rec;
```

If required, defining the template and declaring the structures can be combined. The next example defines an array of structures.

example:

```
struct address_tmp
    {
    char  ad_contact[20];
    char  ad_line_1[30];
    char  ad_line_2[30];
    char  ad_line_3[30];
    char  ad_line_4[30];
    char  ad_phone[15];
    }  company_address[MAX_ADDRESSES];
```

The structure tag is, in fact, optional and so if the structure definition is not required elsewhere it can be omitted. The next example also illustrates that structures can be nested.

example:

```
struct
    {
    char  name[20];
    char  date[8];
    struct  address_tmp  address;
    }  cust_rec;
```

As with unions, the only operations that can be undertaken on structures are taking its address using the & operator, and accessing its individual members using the structure member operator . .

structure_name.member_name

example:

```
write_staff_rec.st_salary = 9500;
```

The only method of passing structures to functions is by means of pointers.

example:

```
process_staff(&write_staff_rec);
. . .
. . .
process_staff(st_rec)
    struct  staff_tmp  *st_rec;
    {
    . . .
    };
```

Since `st_rec` points to the structure, the member `st_name` could be referred to as:

```
(*st_rec).st_name
```

However, pointers to structures are used so frequently that a more convenient notation is available using the → operator.

pointer→member_name

So to refer to `st_name` we could use:

```
st_rec→st_name
```

To find the size of a structure in bytes we can use the operator `sizeof`, which is evaluated at compilation time. The object can be an actual variable, array or structure, a data type like `int` or `char` or a structure definition. With Hisoft C, the sizeof operator will only accept predefined types or a `typedef` name.

sizeof(object)

For example, to allocate space for 100 integers using `calloc` we could use:

```
if ((ptr = calloc(100,sizeof(int))) == NULL)
    {
    printf("\nNO SPACE AVAILABLE");
    return FALSE;
    }
```

The use of structures is illustrated in program 14 which records pupils names and exam marks and then generates a list of pupils in order of proficiency.

Program 14: class positions

```
/***********************
*   CLASS POSITIONS    *
*----------------------*
*    M.R.H.   May 85   *
***********************/

#include <stdio.h>

#define NO_OF_SUBS    8
#define NO_OF_PUPILS 40
#define TRUE          1
#define FALSE         0
#define stdin         0

typedef int *location;

char *sub_text[NO_OF_SUBS] =
   {
   "Maths",
   "English",
   "French",
   "Geography",
   "History",
   "Chemistry",
   "Physics",
   "Biology"
   };
```

```
int max_mark[NO_OF_SUBS], cnt;
unsigned val;

struct class_tmp
   {
   int total;
   char name[20];
   int  mark[NO_OF_SUBS];
   };

struct class_tmp class_rec[NO_OF_PUPILS];

extern int total_comp();

main()
   {
   printf("Calculation of class positions");
   if (get_max_marks())
      return;
   printf("\nEnter CTRL Z to complete input");
   cnt = 0;
   while (get_marks())
      ;
   qsort(class_rec,cnt,sizeof(struct class_tmp),total_comp);
   disp_positions();
   }

get_max_marks()
   {
   int j;

   printf("\n\nEnter maximum possible marks for each subject\n\n");
   for (j = 0; j < NO_OF_SUBS; j++)
      {
      printf("%s: ",sub_text[j]);
      scanf(" %d",&max_mark[j]);
      if (max_mark[j] > 600)
         {
         printf("\nMaximum mark must be between 1 and 600");
         return TRUE;
         }
      }
   getchar();
   return FALSE;
   }

get_marks()
   {
   int j;

   printf("\n\nEnter name and marks for each subject\n");
   printf("\n%d Name: ",cnt + 1);
   fgets(class_rec[cnt].name,20,stdin);
   if (strlen(class_rec[cnt].name) < 2)
      return FALSE;
      class_rec[cnt].total = 0;
   for (j = 0; j < NO_OF_SUBS; j++)
      {
      do
         {
         printf("%s: ",sub_text[j]);
         scanf(" %d",&val);
         }
```

```
    while (val > max_mark[j]);
    class_rec[cnt].mark[j] = val * 100 / max_mark[j];
    class_rec[cnt].total += class_rec[cnt].mark[j];
    }
cnt++;
getchar();
return TRUE;
}

disp_positions()
    {
    int j;

    printf("\nPOS  TOTAL     NAME\n\n");
    for (j = 0; j < cnt; j++)
        {
        printf("%2d     %3d     %s",j + 1,
                        class_rec[j].total,class_rec[j].name);
        }
    }

total_comp(ptr1,ptr2)
    char *ptr1, *ptr2;
    {
    location tot1, tot2;

    tot1 = cast(location) ptr1;
    tot2 = cast(location) ptr2;
    return (*tot2 - *tot1);
    }

#include ?stdio.lib?
```

8.4 Dynamic Data Structures

There are occasions when we require a flexible system of ordering our data, and we now turn to several different methods which can be covered by the general title *dynamic data structures.*

As an example of what dynamic data structures are all about, consider the table in Figure 8.1 which contains a list of customer's orders with a car dealer. All the records are kept in order of the customer's surname which enables the information referring to a particular customer to be located easily. Suppose further that a record for the customer HARRISON has to be added to the list.

So that the table remains in order, the new record must be placed between HART and HOPTON. If these records were kept using normal office index cards, any new record could be added by slotting the new card in at the appropriate position. Likewise, when the customer's order has been completed the corresponding record could be deleted from the system by simply removing the card.

NAME	*INITIAL*	*MODEL*	*COLOUR*	*DATE DUE*
BANKS	M	240 DL	Green	04.5.83
BEECH	G	760 GLE	Silver	30.6.83
GRAHAM	R	360 GLT	Red	25.5.83
HART	W	240 GL	White	25.5.83
HOPTON	J	360 GLS	Blue	17.6.83
TROWSDALE	D	260 GLE	Black	19.6.83
TURNER	M	340 DL	Red	17.6.83
HARRISON	M	340 GL	Silver	30.6.83

Figure 8.1

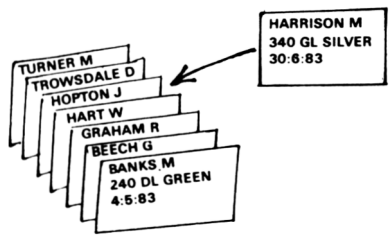


Figure 8.2

Our requirement is to simulate such a system where we can add and delete records, and by re-using any vacated space, keep the occupied memory down to a minimum. One data structure that can be used for this application is called a *linked list*. When using linked lists for small quantities of data, the advantages of flexibility are offset by the disadvantages of complexity. However, it is useful to learn the techniques since general and more complicated structures can then be handled in a similar fashion.

8.5 The Forward Linked List

With linked lists, each element contains two items of information; the first item is the data itself and the second item is a pointer which relates that element to the next. The data can contain several fields, e.g. name, address, telephone number, etc.; but for simplicity, our explanations will consider just a single data field.

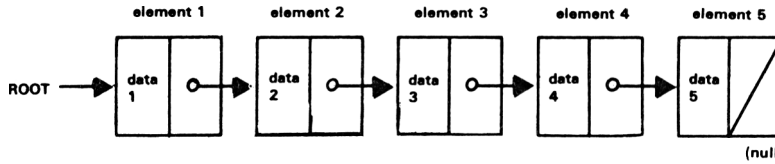


Figure 8.3

In Figure 8.3, element 1 points to element 2, element 2 points to element 3, and so on. As element 5 is the last element in the list its pointer is NULL.

To set up such a list we have to define a self-referential structure.

example:

```
struct link_list
   {
   char data[20];
   struct linked_list *ptr;
   } *root, list[NO_OF_ELEMENTS] ;
```

To retrieve the data we access all the elements in the list; this is done by starting from `root` and then moving along the structure, accessing each item of data, until the NULL pointer is reached.

An item can easily be deleted from a linked list by amending the pointer from the previous element so that it points, instead, to the next item. The pointer of the element that has been deleted should then be changed to a NULL value to signal that it is empty and so available for re-use.

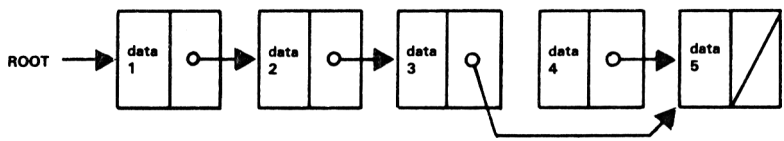


Figure 8.4

If instead we want to reinstate an item into our list there are two stages to undertaken; firstly we must find where in the list the item should be placed, and secondly, amend the pointers to include the new item. The position where the new item is placed depends on what the user wants; it could be simply to put the item immediately after the last item or perhaps, if the list is ordered, in the

correct position as in our example in Figure 8.5.

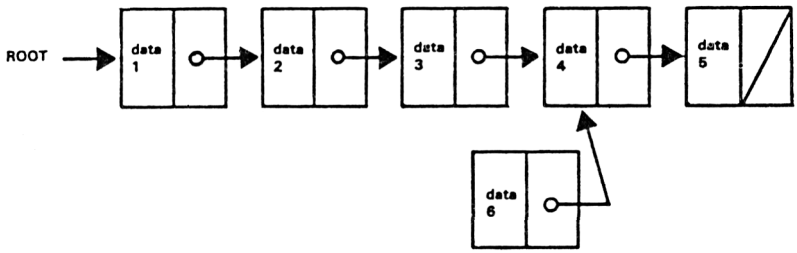


Figure 8.5

First the pointer from element 6 is changed to point to element 4, and then the pointer of element 3 is changed to point to element 6 as in Figure 8.6.

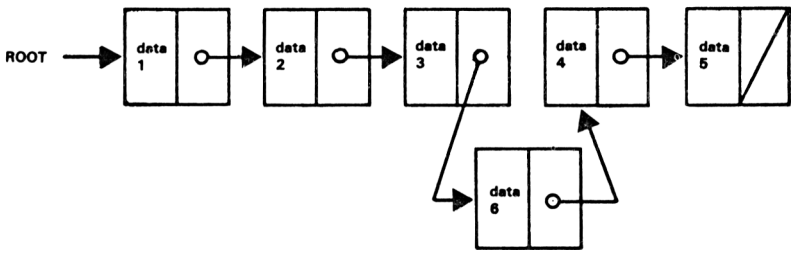


Figure 8.6

We shall now see some routines in program 15 that can handle linked lists; note that special considerations have to be made when the data is to be placed at the beginning or the end of a list.

Program 15: forward linked list

```
/************************
*      LINKED LIST       *
*----------------------*
*    M.R.H.    May 85    *
************************/

#include <stdio.h>
```

```
#define NO_OF_ELEMENTS  100
#define stdin             0
#define TRUE              1
#define FALSE             0

struct link_list
   {
   char data[20];
   struct link_list *ptr;
   } *root, *lp, *lp1, list[NO_OF_ELEMENTS];

char buffer[50];
int x;

main()
   {
  char opt[3];

   while (TRUE)
     {
     printf("\n\nLINKED LISTS");
     printf("\n\n1. Initialise");
     printf("\n2. List");
     printf("\n3. Insert");
     printf("\n4. Delete.");
     printf("\n5. Exit ");
     printf("\n\nSelect facility required: ");
     fgets(opt,3,stdin);
     switch(opt[0])
        {
        case '1':
           init_chain();
           break;
        case '2':
           list_chain();
           break;
        case '3':
           insert_item();
           break;
        case '4':
           delete_item();
```

```
            break;
         case '5':
            return;
         default:
            printf("\nInvalid option");
            bell();
         }
      }
   }

init_chain()
   {
   root = NULL;
   for (x = 0; x < NO_OF_ELEMENTS; x++)
      {
      strcpy(list[x].data,"");
      list[x].ptr = NULL;
      }
   }

list_chain()
   {
   lp = root;
   printf("\nLIST CHAIN\n\n");
   while (lp != NULL)
      {
      printf("%s",lp->data);
      lp = lp->ptr;
      }
   }

insert_item()
   {
   printf("\nINSERT ITEM\n");
   printf("\nEnter data: ");
   fgets(buffer,20,stdin);
   for (x = 0; x < NO_OF_ELEMENTS; x++)
      {
      if (list[x].data[0] == '\000')
         {
         strcpy(list[x].data,buffer);
```

```
        if (root == NULL)
            {
            root = &list[x];
            root->ptr = NULL;
            return;
            }
        if (strcmp(root->data,buffer) > 0)
            {
            list[x].ptr = root;
            root = &list[x];
            return;
            }
        lp = root;
        while (lp != NULL)
            {
            if (strcmp(lp->data,buffer) > 0)
                {
                list[x].ptr = lp;
                lp1->ptr = &list[x];
                return;
                }
            lp1 = lp;
            lp = lp->ptr;
            }
        lp1->ptr = &list[x];
        list[x].ptr = NULL;
        return;
        }
    }
}

delete_item()
    {
    printf("\nDELETE ITEM\n");
    printf("\nEnter data: ");
    fgets(buffer,20,stdin);
    if (root != NULL)
        {
        if (strcmp(root->data,buffer) == 0)
            {
            strcpy(root->data,"");
```

```
            root = root->ptr;
            printf("\nItem deleted");
            return;
            }
        lp = root;
        while (lp != NULL)
            {
            if (strcmp(lp->data,buffer) == 0)
                {
                strcpy(lp->data,"null");
                lp1->ptr = lp->ptr;
                printf("\nItem deleted");
                return;
                }
            lp1 = lp;
            lp = lp->ptr;
            }
        }
    printf("\nItem not in list");
    bell();
    }

#include ?mrh.lib?

#include ?stdio.lib?
```

8.6 More Advanced Lists

Although many applications can be handled by forward linked lists, there are a couple of amendments that can be made which enhance the power of the structures that we have seen.

8.7 Circular Lists

In a *circular list* or *ring* the last element is made to point back to the first element. The main advantage of such a structure is that an item which precedes an identified element, can still be accessed without having to restart at the *root* - see Figure 8.7.

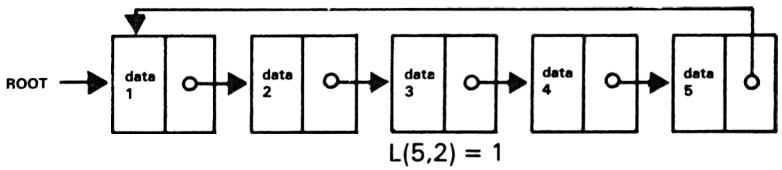


Figure 8.7

8.8 Double Linked Lists

Even greater power can be added to linked lists by including a backward pointer which links up an element with its predecessor in the structure - see Figure 8.8.

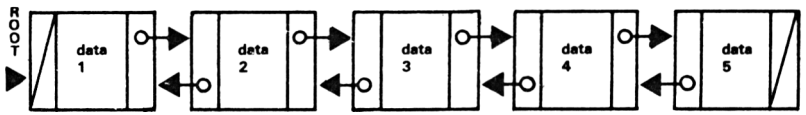


Figure 8.8

This enables the structure to be searched in either direction. By using more than one pointer it is possible to order the list in more than one type of order.

In certain small applications, the advantages of such a structure will be offset by an increase in the memory required for additional pointer storage.

8.9 Stacks and Queues

There are two very useful linear data structures that are commonly used in computing called *stacks* and *queues*.

A stack is a method of storing and retrieving data in the computer with the basic principle that the most recent item of data stored will be the first retrieved. Storing information is known as *pushing* onto the stack and retrieving it is known as *popping* or *pulling* the stack. In our example in Figure 8.9, the top of the structure is indicated by a pointer called `top`.

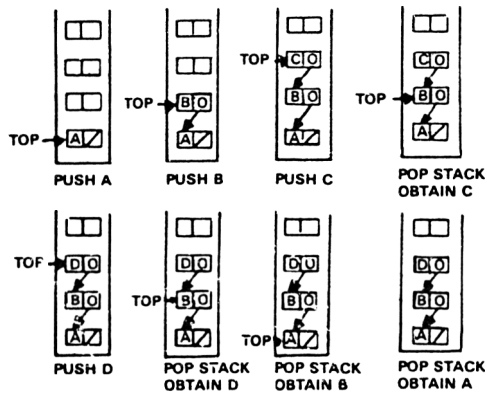


Figure 8.9

Although it is never obvious to the programmer, the C program language uses a stack when dealing with subroutine calls. When a function is called, the calling address is popped onto the stack and control jumps to the called function. Automatic variables are also placed on the stack. When a program returns from a function, it continues from the address that is popped from the stack. This enables functions to be nested several levels deep although careful management is required by the programmer.

A queue structure differs in that its rule is that the earliest item stored will be the first retrieved. Two pointers are required, `head` and `tail` which point in the structure to the first and last items respectively.

A queue is used with keyboard buffers where keys are stored as they are pressed. Since it is a queue, the earliest keys pressed are detected before the more recent keys. Any released space is then available to store future key presses.

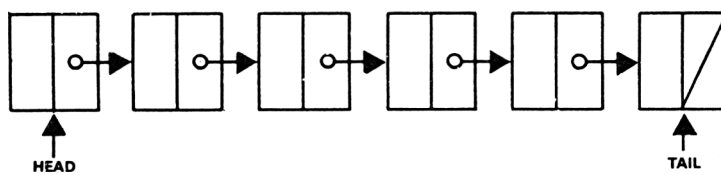


Figure 8.10

When an item is retrieved, `head` is made to point to the next element in the structure, i.e. to the element that is pointed to by the pointer of the element that is being retrieved.

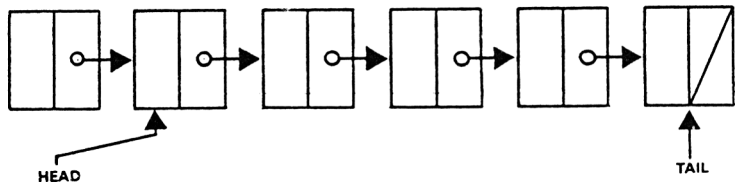


Figure 8.11

When an item is added both `tail` and the last element are made to point at the new item.

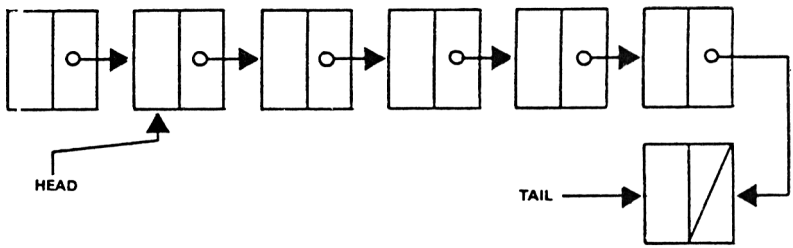


Figure 8.12

The main problem is that a queue gradually drifts through the memory as retrievals and additions occur. One solution is to use a circular list so that if all the memory at the end of a structure runs out, then items can be added at the beginning. If `tail` ever reaches `head` then we have run out of space.

8.10 Graphs

Although the concept of the linked lists is very useful as an insight into data structures, their use is severely limited as they can only function in one dimension, either forwards or backwards. In order to utilise these techniques with any practical ideas we often have to handle data structures in more than one dimension; such structures are called *graphs*.

Consider Figure 8.13 which represents the air routes that a certain airline company undertakes.

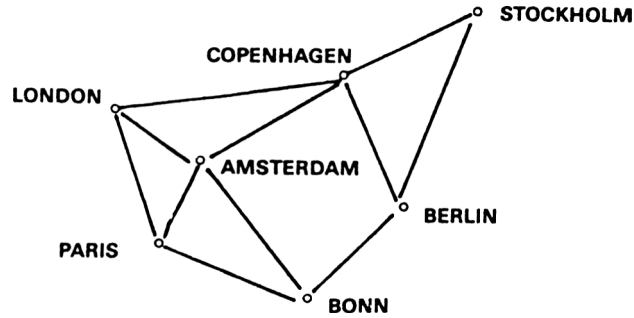


Figure 8.13

This could be represented by the structure in Figure 8.14.

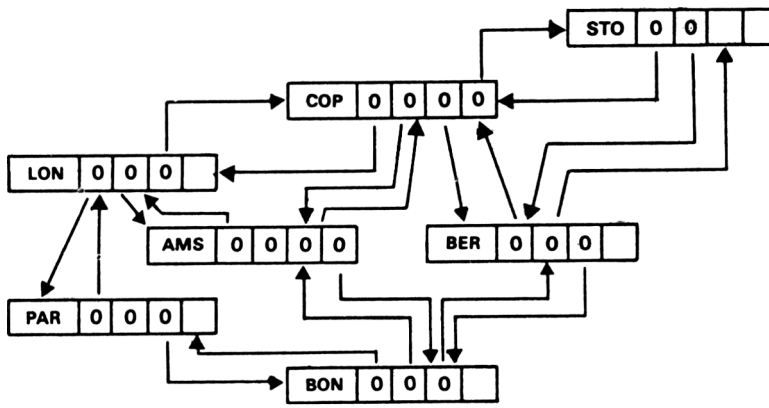


Figure 8.14

Additional data items could be included which, for example, could give the cost, distance, etc. between two cities.

When referring to graphs, the intersecting points are called *nodes* and the links

that join the nodes are called *edges*. Edges can be either directed or undirected, and may or may not contain a value (for instance, cost, distance, etc., in our above example). Nodes can represent many things; such as towns or positions, production output, steps in a process, etc.

Graphs have many applications; as an example we shall take a look at the problem of finding the shortest path between two nodes. With reference to Figure 8.13, what is the shortest route from London to Berlin.

The algorithm that we shall use calculates the shortest paths between all pairs of nodes in a given graph, and can also print out the optimum route.

The graph of n nodes may be represented in the computer by a n x n array (or matrix), say `x(n,n)` where element `x(s,d)` would be the distance from a source town `s` to a destination town `d` - see Figure 8.15.

e.g.

	1	2	3	4	5	6
1	0	8	∞	∞	7	∞
2	8	0	15	∞	10	10
3	∞	15	0	10	∞	5
4	∞	∞	10	0	10	7
5	7	10	∞	10	0	∞
6	∞	10	5	7	∞	0

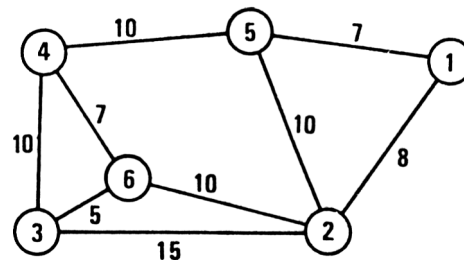


Figure 8.15

Any two towns that are not directly connected take a distance value of infinity - with Hisoft C we have to make do with a value of 65535.

The procedure for calculating the shortest route is as follows:

The distance of each route is considered separately with the distance of the same route but going through each other node (if possible). For example, distance 1 → 6 is compared to the distances 1 → 2 → 6, 1 → 3 → 6, 1 → 4 → 6 and 1 → 5 → 6, and the shortest value is placed into our array at element x(1,6). A separate array is used to record the optimum route.

Shortest Routes

Program 16 allows you to enter a graph, edge by edge by specifying a source

town number, a destination town number and a distance. When you have entered all the information, you simply input a set of invalid data, such as 0,0,0. The program then works out the shortest route between each pair of towns. Then by specifying two town numbers, the program will display the shortest route and its distance.

Program 16:

```
/***********************
*   SHORTEST ROUTES     *
*-----------------------*
*    M.R.H.   May 85    *
***********************/

#define NO_TOWNS       10
#define INFINITY    65535
#define TRUE            1

unsigned x[NO_TOWNS][NO_TOWNS], y[NO_TOWNS][NO_TOWNS];
unsigned s, d, v;

main()
   {
   printf("\nSHORTEST ROUTES\n");
   initialise();
   get_distances();
   calc_distances();
   access_distances();
   }

initialise()
   {
   int p, q;

   for (p = 0; p < NO_TOWNS; p++)
      for (q = 0; q < NO_TOWNS; q++)
         if (p != q)
            x[p][q] = INFINITY;
   }

get_distances()
   {
   while (TRUE)
      {
      printf("\nEnter source town      : ");
      scanf(" %d",&s);
      printf("Enter destination town: ");
      scanf(" %d",&d);
      printf("Enter distance:        : ");
      scanf(" %d",&v);
      if (s == d || s < 1 || s > NO_TOWNS || d < 1 || d > NO_TOWNS)
         return;
      else
         {
         x[s-1][d-1] = v;
         x[d-1][s-1] = v;
         y[d-1][s-1] = s;
         y[s-1][d-1] = d;
         }
      }
   }
```

```
calc_distances()
   {
   int p, q, r;
   unsigned k;

   printf("\nPlease wait\n");
   for (p = 0; p < NO_TOWNS; p++)
      for (q = 0; q < NO_TOWNS; q++)
         for (r = 0; r < NO_TOWNS; r++)
            {
            if (x[q][p] == INFINITY || x[p][r] == INFINITY)
               k = INFINITY;
            else
               k = x[q][p] + x[p][r];
            if (x[q][r] > k)
               {
               x[q][r] = k;
               y[q][r] = y[q][p];
               }
            }
   }

access_distances()
   {
   printf("\nLowest cost between two towns");
   while (TRUE)
      {
      printf("\n\nEnter first town : ");
      scanf(" %d",&s);
      printf("Enter second town: ");
      scanf(" %d",&d);
      if (s >= 1 && s <= NO_TOWNS && d>= 1 && d <= NO_TOWNS)
         {
         if (x[s-1][d-1] == INFINITY || x[s-1][d-1] == 0)
            printf("Not connected");
         else
            {
            printf("Cost from town %d to town %d is %d",
                              s,d,x[s-1][d-1]);
            printf("\nVia towns - ");
            if (y[s-1][d-1] == d)
               printf("Direct");
            else
               {
               s = y[s-1][d-1];
               while (s != d)
                  {
                  printf(" %d \,",s);
                  s = y[s-1][d-1];
                  }
               }
            }
         }
      else
         return;
      }
   }
```

8.11 Trees

There is one special graph commonly used in computing called a *tree*. A tree is a graph that contains no isolated nodes and no cycles, that is, there is one and only one route for getting from one node to another. We are usually concerned with trees in which all the nodes are directed away from one specific node called the *root*; every node except for the root has exactly one edge entering it.

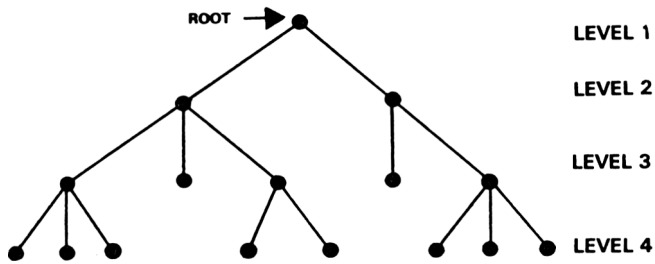


Figure 8.16

The number of nodes away from the root is often referred to as the tree's level (with root situated at level 1). The depth of the tree is its maximum level. Often a preceding node is called a parent node and a descending node is called a sibling.

In order to keep things simple, we shall restrict ourselves to *binary trees*; these have a maximum of two edges leading from each node.

Binary trees may be stored in C using a similar principle to linked lists, except that two pointers are required for each structure.

example

```
struct tree_node
   {
   char data[20];
   struct tree_node *left_ptr;
   struct tree_node *right_ptr;
   } *root, tree[NO_OF_NODES];
```

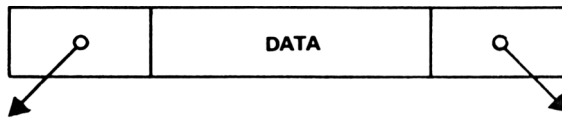


Figure 8.17

As with linked lists we require the data to be set up in some logical order. For example, let us see how we would store the following sequence of car manufacturers in alphabetical order in a binary tree.

MERCEDES,FERRARI,PORSCHE,LOTUS,VOLVO,BMW,SAAB.

We commence with the first name, MERCEDES, as the root of the tree; at this stage the node has no descendants so the pointers take NULL values.

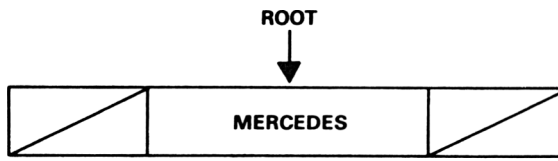


Figure 8.18

Next we introduce FERRARI and since it precedes MERCEDES alphabetically it will become a descendant on the left.

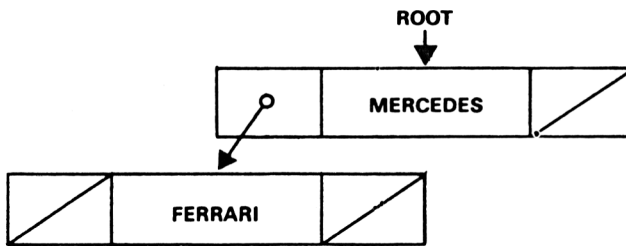


Figure 8.19

The next item, PORSCHE follows MERCEDES and so becomes a right descendant of it.

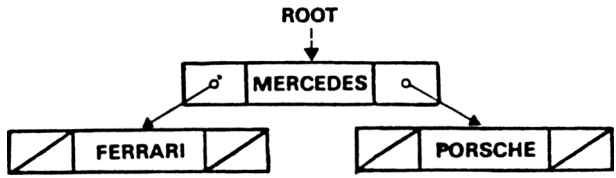


Figure 8.20

After examining all the information and arranging it alphabetically the tree resembles Figure 8.21.

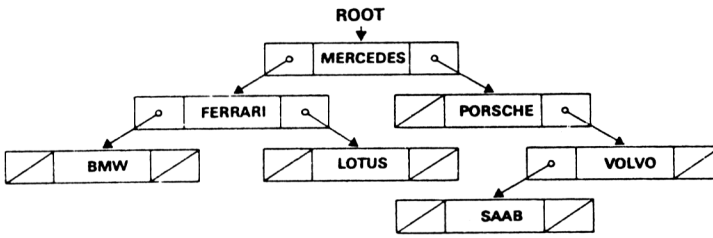


Figure 8.21

The next obvious task is to be able to select a name and to request all the data that is associated with it. Having entered a name, the program searches down the tree starting at the root and then travelling either left or right depending on the alphabetical order of the entered name and the one stored at the present node. If a NULL pointer is reached before the entered name is located then the name is not present in the tree. This method of searching is a form of what is called a *binary chop* and can be very fast, even with vast quantities of data. For example, by making seven comparisons we can get down to level 8 of the tree which could mean that up to 255 elements have been searched.

To output all the data in the tree in the order set up, we require a systematic method of travelling to each node once and in the order of the leftmost nodes before those to their right.

If we consider any node in the binary tree, providing its pointers are not NULL, it is linked to two binary subtrees. Similarly, all the nodes in each of these subtrees are joined to further subtrees. Thus a binary subtree may be abbreviated to the layout shown in Figure 8.22.

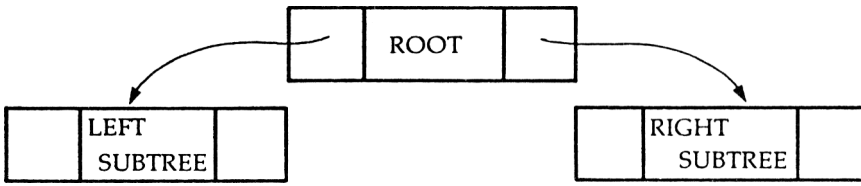


Figure 8.22

If we travel to each of these components in a fixed order it is called a *traversal*

There are three traversals that we can make:

1) Left subtree		1) Left subtree		1) Node
2) Node	or	2) Right subtree	or	2) Left subtree
3) Right subtree		3) Node		3) Right subtree
(LNR)		(LRN)		(NLR)

Further traversals using the same fixed order should be made at each subtree. If we use the order LNR we can obtain our data in the order in which the structure was set up (alphabetically).

Consider the tree in Figure 8.23 under a LNR traversal.

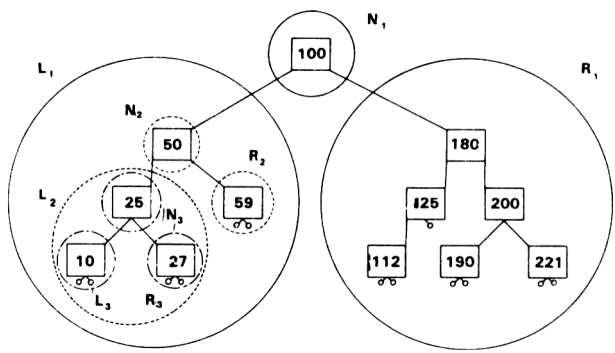


Figure 8.23

Our order would be: L_1, 100, R_1 Subtree L_1 can be traversed LNR to give: L_2, 50, 59 and L_2 can be traversed LNR to give: 10, 25, 27 Thus our order so far is: 10, 25, 27, 50, 59, 100, R_1 When R_1 has been traversed LNR we will have accessed the whole tree in order.

Tree sort

Program 17 demonstrates how to sort alphabetically a sequence of strings by creating a binary tree and then undertaking a LNR traversal.

The traversal is done by starting at the root and continually travelling left pushing the nodes on to a stack. Wherever a NULL pointer is reached, the data value of the node is printed and if a a right hand node is present the same process is taken on its subtree. Having gone as far as possible, we then back-track to the parent node by popping the stack. This process is repeated for the whole tree.

The program simplifies the problem of stack control by making use of recursive functions, that is ones that call themselves.

Program 17:

```
/***********************
*       TREE SORT       *
*-----------------------*
*    M.R.H.    May 85   *
***********************/

#include <stdio.h>

#define NO_NODES     100
#define LINEFEED   '\012'
#define stdin          0

typedef struct tree_node
   {
   char data[20];
   struct tree_node *left_ptr;
   struct tree_node *right_ptr;
   } TREENODE, *TREEPTR;

TREENODE t_node[NO_NODES];
TREEPTR  t_root, tp1, tp;

char buffer[20];
int x;

main()
   {
   init_tree();
   printf("TREE SORT");
   printf("\nEnter data\n");
   get_data();
   printf("\nSorted list\n");
```

```
    list_tree(t_root);
    }

init_tree()
    {
    for (x = 0; x < NO_NODES; x++)
        {
        t_node[x].data[0] = '\000';
        t_node[x].left_ptr = NULL;
        t_node[x].right_ptr = NULL;
        }
    }

get_data()
    {

    while (TRUE)
        {
        printf("; ");
        fgets(buffer,20,stdin);
        if (buffer[0] == LINEFEED)
            return;
        for (x = 0; x < NO_NODES; x++)
            {
            if (t_node[x].data[0] == NULL)
                {
                if (t_root == NULL)
                    {
                    t_root = &t_node[x];
                    strcpy(t_node[x].data,buffer);
                    t_root->left_ptr = NULL;
                    t_root->right_ptr = NULL;
                    break;
                    }
                else
                    {
                    tp = t_root;
                    while (tp != NULL)
                        {
                        tp1 = tp;
                        if (strcmp(cast(char)tp->data,buffer) > 0)
                            tp = tp->left_ptr;
                        else
                            tp = tp->right_ptr;
                        }
                    if (strcmp(cast(char)tp1->data,buffer) > 0)
                        tp = tp1->left_ptr = &t_node[x];
                    else
                        tp = tp1->right_ptr = &t_node[x];
                    strcpy(t_node[x].data,buffer);
                    tp->left_ptr = NULL;
                    tp->right_ptr = NULL;
                    break;
                    }
                }
            }
        }
    }

list_tree(t_ptr)
    TREEPTR t_ptr;
    {
    if (t_ptr->left_ptr != NULL)
```

```
      list_tree(t_ptr->left_ptr);
   printf("%s",cast(char)t_ptr->data);
   if (t_ptr->right_ptr != NULL)
      list_tree(t_ptr->right_ptr);
   }

#include ?stdio.lib?
```

8.12 Heuristic Programming

By now you will be aware of the flexibility that dynamic data structures can provide. Trees are the most common form found in computing and are used in a varied range of unusual subjects. Many games can be studied using trees, where different branches provide the various options available; by using some optimizing algorithms a computer can examine all the possible states that could occur for any feasible move it makes.

A similar idea is used when a program learns from previous operations, and remembers what results occur for certain moves by building onto a tree. The program commences with a tree consisting of a single node, the root, and at that stage knows nothing; with time the tree expands and soon becomes "intelligent". Programs that learn with experience are called *heuristic*.

Program 18: animals

Heuristic programming is demonstrated in Program 18 which starts off with minimal knowledge of animals. It asks you to think of an animal and then tries to guess it by asking you a series of questions to which you must answer "yes' or "no". If the computer is unable to guess the animal it will ask you for a question and answer that it can use in the future.

The program builds up a binary tree with animals stored at the bottom nodes in the tree and questions in the other nodes. By asking questions and analysing the answers the program can decide on which path to travel down the tree. When an animal is reached, the computer asks if it is the correct one; if not, it asks for a question that distinguishes between the animal in the tree and the animal chosen.

There is one small snag; when you turn off your computer, it will forget all that it has ever learned about frogs, elephants and whatever. This is rectified by including a facility for the program to load and save the data on disk.

```
/***********************
*       ANIMALS        *
*----------------------*
*    M.R.H.   May 85   *
***********************/

#include <stdio.h>

typedef struct tree_node
   {
   char data[30];
   struct tree_node *left_ptr;
   struct tree_node *right_ptr;
   } TREENODE, *TREEPTR;

TREEPTR t_root, t_ptr;
extern TREEPTR talloc(), read_node();

char animal[30], question[30];

FILE *fp;

main()
   {
   printf("ANIMALS\n\n");
   if (initialise())
      {
      while (TRUE)
         {
         process();
         printf("\nPlay again ? ");
         if (!get_yn())
            {
            printf("\nDo you want to update disk ? ");
            if (get_yn())
               write_file();
            return;
            }
         }
      }
   }

initialise()
   {
   int ret;

   printf("Initialising animal tree\n");
   printf("Do you want to read animals from disk ? ");
   if (get_yn())
      {
      if (ret = read_file())
         return ret;
      }
   else
      default_tree();
   return TRUE;
   }
```

```
read_file()
   {
   if ((fp = fopen("ANIMAL.DAT","r")) == NULL)
      {
      printf("\nFailure opening ANIMAL.DAT\n");
      return FALSE;
      }
   t_root = read_node();
   fclose(fp);
   return TRUE;
   }

TREEPTR read_node()
   {
   TREEPTR tmp;
   char buffer[30];
   int k;

   for (k = 0; k < 30; k++)
      buffer[k] = getc(fp);
   if (buffer[0] != '\000')
      {
      tmp = talloc();
      strcpy(tmp->data,buffer);
      tmp->left_ptr = read_node();
      tmp->right_ptr = read_node();
      return tmp;
      }
   else
      return NULL;
   }

default_tree()
   {
   printf("\nAssigning default tree\n");
   t_root = talloc();
   strcpy(cast(char)t_root->data,"Is it a bird");
   t_root->left_ptr = talloc();
   t_root->right_ptr = talloc();
   strcpy(((t_root->left_ptr)->data),"ROBIN");
   (t_root->left_ptr)->left_ptr = NULL;
   (t_root->left_ptr)->right_ptr = NULL;
   strcpy(((t_root->right_ptr)->data),"SQUIRREL");
   (t_root->right_ptr)->left_ptr = NULL;
   (t_root->right_ptr)->right_ptr = NULL;
   }
```

```
process()
   {
   printf("\n\nThink of an animal\n");
   delay(10000);
   printf("\nAnswer Y or N to the following questions:\n");
   t_ptr = t_root;
   while (t_ptr->left_ptr != NULL || t_ptr->right_ptr != NULL)
      {
      printf("\n%s ? ",t_ptr->data);
      if (get_yn())
         t_ptr = t_ptr->left_ptr;
      else
         t_ptr = t_ptr->right_ptr;
      }
   printf("\nI think your animal is %s ",t_ptr->data);
   printf("\nAm I correct ? ");
   if (get_yn())
      printf("\nI thought so");
   else
      {
      printf("\nWhat had you chosen ?\n");
      get_string(animal,30);
      printf("Enter a question that distinguishes a %s from a %s \n"
                  animal,t_ptr->data);
      get_string(question,30);
      t_ptr->left_ptr = talloc();
      t_ptr->right_ptr = talloc();
      printf("Would the answer for %s be Y or N ? ",animal);
      if (get_yn())
         {
         strcpy( (t_ptr->left_ptr)->data, animal);
         strcpy( (t_ptr->right_ptr)->data, t_ptr->data);
         }
      else
         {
         strcpy( (t_ptr->left_ptr)->data, t_ptr->data);
         strcpy( (t_ptr->right_ptr)->data, animal);
         }
      strcpy(t_ptr->data,question);
      (t_ptr->left_ptr)->left_ptr = NULL;
      (t_ptr->left_ptr)->right_ptr = NULL;
      (t_ptr->right_ptr)->left_ptr = NULL;
      (t_ptr->right_ptr)->right_ptr = NULL;
      }
   }

write_file()
   {
   if ((fp = fopen("ANIMAL.DAT","w")) == NULL)
      {
      printf("\nFailure opening ANIMAL.DAT\n");
      return FALSE;
      }
   write_node(t_root);
   fclose(fp);
   }
```

```
write_node(tp)
   TREEPTR tp;
   {
   int k;

   if (tp == NULL)
      {
      for (k = 0; k < 30; k++)
         putc('\000',fp);
      return;
      }
   else
      for (k = 0; k < 30; k++)
         putc(*(tp->data + k),fp);
   write_node(tp->left_ptr);
   write_node(tp->right_ptr);
   }

TREEPTR talloc()
   {
   extern char *calloc();
   char *ptr;

   ptr =  calloc(1,sizeof(TREENODE));
   if (ptr == NULL)
      {
      printf("\nSystem failure: Increase size of heap\n");
      getchar();
      }
   return (cast(TREEPTR) ptr);
   }

delay(duration)    /* MRH.LIB */
   int duration;
   {
   while (duration--)
      ;
   }

get_yn()           /* MRH.LIB */
   {
   char opt;

   for ( ; ; )
      {
      switch (rawin())
         {
         case 'y':
         case 'Y':
            printf("Y");
            return TRUE;
         case 'n':
         case 'N':
            printf("N");
            return FALSE;
         default:
         }
      }
   }
```

```
get_string(ptr,length)          /*  MRH.LIB  */
   char *ptr;
   int length;
   {
   char buffer[80];

   while (TRUE)
      {
      gets(buffer);
      if (strlen(buffer) < length)
         {
         strcpy(ptr,buffer);
         return;
        }
      else
         printf("\nRe-enter, max length exceeded\n");
      }
   }

#include ?stdio.lib?
```

Since the topic of data structures is both large and complicated, anyone interested in further study should obtain one of the numerous books specialising in the subject. A recommended book is "*Successful Software for Small Computers*" by Graham Beech, published by Sigma Press, which contains numerous examples of various data structures.

CHAPTER 9

ADVANCED INPUT/OUTPUT TECHNIQUES

9.1 Introduction

In this chapter we shall discuss I/O techniques with C and look at methods for improving the relationship between man and machine, sometimes called *the interface*.

A programmer often puts a great deal of effort into improving the efficiency of a program whereas time spent on the ease of input for the user, eliminating the entry of invalid data, keeping the user fully aware of a program's state, and the presentation of results would be far more beneficial. Information may be output directly to the user by three distinct methods; shape, colours and sounds, each of which is important in its own way.

Shapes can take the form of pictures or symbols which can be recognised immediately, or text, which although it has to be deciphered should not pose too much of a problem. Whilst to the user text is less direct than symbols, it can often say much more in an efficient way. A further extension to shapes is animation in which the screen is updated rapidly to produce a dynamic display. Such displays are sensed quickly by the user; for example, a flashing error message has more effect than a static one. Dynamic displays can provide realistic images and are popular in arcade-type games.

Colour is less powerful than shape but can be used to aid the identification of indistinguishable shapes. For example, a colour display with the top half in blue and the bottom half in green would resemble sky and grass far better than a monochromatic display could. Colour can also be used to highlight certain areas of the screen; for example, a warning message displayed in a vivid red would soon be brought to the attention of the user. Finally, a colour display is nicer to look at than a two tone display, provided that the colours are chosen wisely.

Whilst the user must watch the screen to receive information in shape and colour form, sound output can reach the user even when he/she is not paying direct attention. Thus sound can be useful for attracting the attention of the user, if for example an operation is complete or an error has occurred. Such notes should be high and pleasant for correct operations and long low dirges for the occurrence of errors. Sound can also add reality to a display. In a space invader game, zapping and explosions sounds add realism as well as signalling to the player the results of his operations.

One major cause of user dissatisfaction is the long delays that occur in programs with long detailed numerical calculations. If programs are as efficient as possible, then lengthy inactive screens with the computer showing no sign of life should be avoided by showing screen messages such as "Program initiated", "Stage one complete", etc. This will not only reassure the user but also keep him/her informed of the processing undertaken. At the end of a long process it is probably advisable to attract the user's attention with a bleep type sound.

The user may enter data directly into the computer by using either the keyboard or a games controller. The keyboard is by far the most flexible and powerful means for allowing entry of text, numerical values or single key responses. Most modern keyboards have over 75 keys, but each has alternative meanings when pressed simultaneously with the SHIFT or CTRL keys. Unfortunately it is common for a programmer who uses a keyboard frequently to forget that the users of their programs might not feel equally at ease. In this chapter we shall see how to make the entry of data via the keyboard as simple as possible. Whenever a user has to enter data, clear prompts should be given and any invalid data should be rejected with an error message explaining the problem; the opportunity should be given for the user to re-enter the data. Since keys have predefined labels it makes sense to try to make use of them as much as possible. For example, the best key for ending a program might be CTRL X (eXit) or CTRL Q (Quit), whereas to use something like CTRL 4 would be, to me, quite meaningless.

An alternative to the keyboard is a games controller, such as a *joystick* or *paddle*; these have severe limitations but can be preferable for some applications. A joystick usually has five or six switches. Each of the first four are set when a knob is pushed left, right, up or down; diagonal movements cause two switches to be set. The remaining switches are set when separate push buttons are pressed. A paddle is similar to the volume control on a radio and enters a single value, within a limited range, into the computer. However, the real advantage of a games controller is that the interface to the machine becomes second nature; after a few minutes of moving the joystick to the left or right in order to move an object on the screen left or right you start to do it without giving it any thought.

We shall now proceed to study some I/O techniques but remember the points we have covered in this introduction, notably:

1. Ease and validation of input.
2. Keeping the user fully aware of all operations.
3. Presentation of results.

9.2 Screen Input

The most obvious and commonly used method for entering data is with either the `scanf`, `getchar` or `fgets` standard functions which we met in Chapter 7.

Recapping, `scanf` can be used to input several variables that match a control string. They can be entered simply by separating them with white space characters and the function returns the number of values that successfully matched the control string and were assigned.

`getchar` is a special case of the `getc` function and inputs a single character from the keyboard buffer. Input via `getchar` is *buffered* – when the function is called characters can be input into the keyboard buffer until the Enter key is pressed. The function returns the first character input. Further calls to the function return the next keys in the buffer, or if empty, will wait and accept more characters until Enter is pressed once again. All input characters are echoed on the screen and may be edited using the Delete key.

`fgets` is used to input a sequence of characters from a file and is terminated either by a Newline character or when a maximum count is exceeded. To input from the keyboard the standard input file is used. Both a Newline and a Null character will be present at the end of the entered character sequence.

9.3 Raw Input

Most C compilers will have a function enabling the input of characters without displaying the cursor or any characters on the screen. This is useful for special applications like games. We shall use this facility later in our screen handler where we will input the characters, one by one, using this method and only display them if they are valid. The function is often supplied with another one which detects whether any keys have been pressed.

With Hisoft C these functions are called `rawin` and `keyhit`, and are both

built-in; users of other versions should consult their documentation to find the names of their equivalent functions.

If `rawin` is called and no key is pressed, the program will stop for input. In some cases, for example in arcade type games, it may be required to take some certain actions if there is no key input. The use of `keyhit` and `rawin` together is shown below.

example:

```
if (keyhit())
    c = toupper(rawin());
```

Once a key has been pressed, `keyhit` will continue to return True, until an input character has been read.

The next program concatenates the characters input using these functions to force the user to enter a date string of the correct format, and is a good alternative to the `scanf` function. Once entered, the complete string can be checked for validity.

Program 19:

```
/**********************
*       GET DATE       *
*----------------------*
*     MRH    MAY 85    *
**********************/

#include <stdio.h>


#define TRUE        1
#define FALSE       0

char date[9];

main()
   {
   do
      get_date("Please enter todays date",date);
   while (!validate_date(date));
   }

get_date(prompt_ptr,date_ptr)        /* MRH.LIB */
   char *prompt_ptr, *date_ptr;
   {
   int j;

   printf("%s: ",prompt_ptr);
   for (j = 0; j < 8; j++)
      {
      if (j == 2 || j == 5)
         *(date_ptr + j) = ':';
      else
```

```
            *(date_ptr + j) = rawin();
         printf("%c",*(date_ptr + j));
         }
   }

validate_date(date_ptr)                    /* MRH.LIB */
   char *date_ptr;
   {
   int d, m ,y;

   d = atoi(date_ptr);
   m = atoi(date_ptr + 3);
   y = atoi(date_ptr + 6);
   if (y < 80 || y > 99)
      {
      printf("\nYear outside system range\n");
      bell();
      return FALSE;
      }
   if (m < 1 || m > 12)
      {
      printf("\nInvalid month\n");
      bell();
      return FALSE;
      }
   if (m == 2)
      if (y % 4)
         {
         if (d < 1 || d > 28)
            {
            printf("\nInvalid day\n");
            bell();
            return FALSE;
            }
         }
      else
         {
         if (d < 1 || d > 29)
            {
            printf("\nInvalid day\n");
            bell();
            return FALSE;
            }
         }
   if (m == 4 || m == 6 || m == 9 || m == 11)
      {
      if (d < 1 || d > 30)
         {
         printf("\nInvalid day\n");
         bell();
         return FALSE;
         }
      }
   else
      {
      if (d < 1 || d > 31)
         {
         printf("\nInvalid day\n");
         bell();
         return FALSE;
         }
      }
   return TRUE;
```

```
#include ?mrh.lib?

#include ?stdio.lib?
```

9.4 Validity of Data

One method of preventing programs from crashing is to check that the data being entered is of the correct form. If the data is not correct then control should return to the same input statement. The program should only continue if data of the correct format is entered. It is a good idea to display a message to indicate to the user what the problem was.

Example:

```
prompt_yn(ptr)
    char *ptr;
    {
    printf("%s",ptr);
    while (TRUE)
        {
        while (!keyhit())
            ;
        c = toupper(rawin());
        switch (c)
            {
            case "N":
                return FALSE;
            case "Y":
                return TRUE;
            default:
                error_msg("Invalid response");
            }
        }
    }
```

We saw earlier how to force the user to enter data in the correct format, in our example a date string, but this does not guarantee that the entered values are valid, i.e. a month of 13 could have been given. It is, however, a simple task to then check that the month lies between 1 and 12, and then by referring to a list of days in each month that the date is valid - of course leap years must be taken into account. The numeric values are extracted from a date string using the standard `atof` function. An example of a function for validating a string of the format DD/MM/YY is listed with program 19 above.

Other methods for checking the validity of data include inspecting the lengths

of strings using `strlen`, checking that values are in a given range using the '<'and '>'operators, and checking that certain characters are present by inspecting the ASCII character codes.

Another problem that can occur is when the user enters valid but incorrect input data. If this creates results that are difficult to rectify it is a good idea to re-display the entered data and ask the user to confirm it by pressing Y or N before futher actions are undertaken.

Carelessness in checking input data can result in unnecessary time wasting which, in turn, can cause frustration for the user, so always try to make your programs as robust as possible.

9.5 Screen Output

You will be well aware that output is directed to the screen by using the functions `printf` and `putchar`. Hisoft supply an additional function called `rawout` which acts like `putchar` but avoids some problems which occur due to the way the Amstrad screen interprets certain control characters.

In command mode it is possible to move the cursor about the screen using the cursor control keys. With C we can program these standard ASCII characters into statements using the `printf` functions giving us full cursor control.

example:

```
#define CURS_L '\010' /* cursor left */
#define CURS_R '\011' /* cursor right */
#define CURS_D '\012' /* cursor down */
#define CURS_U '\013' /* cursor_up */
```

The usual procedure to output a single character is to call the function `putc`. However, with the Amstrad, wierd and surprising effects can occur when certain control characters are printed since the C world uses a different convention to Amstrad when interpreting such characters. The solution is to call the Hisoft function `rawout` which will print exactly what is required.

This is illustrated in program 20 which will print five 'V's in a V formation. The program also introduces an alternative syntax for expressing `if .. then .. else` using the ternary operator `? :`.

(expression1) ? (expression2) : (expression3)

expression1 is evaluated; if it is non-zero (TRUE) expression2 is evaluated,

whilst if it is zero (FALSE) expression3 is evaluated. This mechanism is very convenient for conditional parameters to functions as required in program 20.

Program 20:

```
/*********************************
*    CURSOR CONTROL CHARACTERS   *
*--------------------------------*
*          MRH   May 85          *
*********************************/

#define CURS_L '\010'
#define CURS_R '\011'
#define CURS_D '\012'
#define CURS_U '\013'

main()
    {
    int x;

    for (x = 0; x < 5; x++)
        {
        rawout(CURS_R);
        rawout((x < 3) ? CURS_D : CURS_U);
        rawout('V');
        }
    while (!keyhit())
        ;
    }
```

One other useful standard ASCII character to include in your repertoire is 7.

example:

```
#define BELL    '\007'

bell() /* MRH.LIB */
    {
    printf("%c",BELL);
    }
```

9.6 Animated Effects

With the aid of control characters and C control loops we can display a sequence of characters to produce simple animation. The idea is to construct a string which includes the characters to be displayed along with some cursor control characters to move the characters from their previous print position and some blank characters to overwrite the previous display. By controlling the program execution with a loop we can print the string several times in succession to produce some interesting effects.

Program 21: horizontal motion

```
/*************************
*   HORIZONTAL MOTION    *
*-----------------------*
*      MRH  May 85       *
*************************/

#define CURS_L '\010'
#define CURS_R '\011'
#define CURS_D '\012'
#define CURS_U '\013'
#define CLS    '\f'
#define SPACE  ' '

main()
   {
   int j;

   rawout(CLS);
   for ( ; ; )
      {
      for (j = 1; j < 75; j++)
         {
         rawout(SPACE);
         rawout('>');
         rawout(CURS_L);
         delay(250);
         }
      for (j = 1; j < 75; j++)
         {
```

```
            rawout(SPACE);
            rawout(CURS_L);
            rawout(CURS_L);
            rawout('<');
            rawout(CURS_L);
            delay(250);
            }
        }
    }

#include ?mrh.lib?
```

We could use a similar technique to achieve vertical movement. The next example combines both horizontal and vertical movement to produce a diagonal motion.

Program 22: diagonal motion

```
/*************************
*     DIAGONAL MOTION     *
*-----------------------*
*       MRH  May 85       *
*************************/

#define CURS_L '\010'
#define CURS_R '\011'
#define CURS_D '\012'
#define CURS_U '\013'
#define CLS    '\f'
#define SPACE  ' '

main()
    {
    int j;

    for ( ; ; )
        {
        rawout(CLS);
        for (j = 1; j < 20; j++)
            {
            rawout(SPACE);
```

```
            rawout(CURS_D);
            rawout(214);
            rawout(CURS_U);
            rawout(CURS_U);
            rawout(SPACE);
            rawout(CURS_D);
            rawout(214);
            rawout(CURS_L);
            rawout(CURS_L);
            rawout(CURS_L);
            rawout(CURS_D);
            delay(200);
            }
        }
    }

#include ?mrh.lib?
```

9.7 Controlled Printing

The standard method of printing on the screen is either along a line or down a column. This is perfectly acceptable for small quantities of data, but when printing has reached the bottom row, the screen will scroll upwards. Unfortunately, this means that the top of the display will vanish before you have time to read the results!

Most software applications require controlled printing, where output can be positioned anywhere on the screen by having full cursor control. Every system will have a method of controlling the position of the cursor by sending a sequence of characters followed by the screen coordinates of the position required. Unfortunately, there is no standard for this cursor control sequence and so you will have to consult your system documentation. The following example positions the cursor to (x,y) on the Amstrad.

example:

```
cursor_pos(x,y)  /* MRH.LIB */
    char x,y;
    {
    rawout(31);
    rawout(y);
    rawout(x);
    }
```

Sending other sequences of characters in this fashion can have other effects - such as changing the number of characters per line or the screen colours. Again consult your documentation to see how to change screen modes, colour and other possible screen effects. With the Amstrad:

example:

```
screen_mode(m)      /* MRH.LIB */
    char m;
    {
    rawout(4);
    rawout(m);
    }

clear_screen()      /* MRH.LIB */
    {
    rawout("\f");
    }

paper(ink)      /* MRH.LIB */
    char ink;
    {
    rawout(14);
    rawout(ink);
    }

pen(ink)      /* MRH.LIB */
    char ink;
    {
    rawout(15);
    rawout(ink);
    }

ink(ink,col1,col2)      /* MRH.LIB */
    char ink,col1,col2;
    {
    rawout(28);
    rawout(ink);
    rawout(col1);
    rawout(col2);
    }
```

9.8 Menu Selection

We saw in Chapter 3 how we should structure our programs by breaking them down into blocks or modules, each of which has a specific purpose. An ideal method of producing a program that undertakes several functions is to write each in a separate block of code and then allow the user to select the required section by means of a menu displaying the various available options as in Figure 9.1.

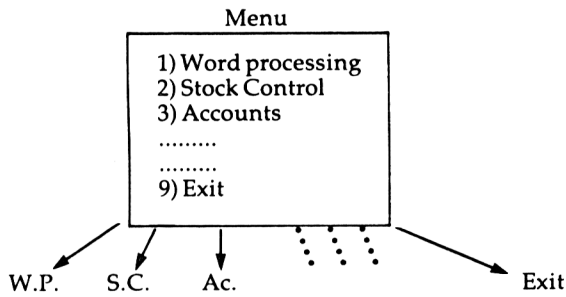


Figure 9.1

Of course there is no reason why selecting a certain option could not lead to another menu displaying a further list of related options so that the user has to make an additional selection or return back to the main menu.

There are numerous ways in which a menu selection program can be written. The most common is the *Chinese take-away* method where the user selects a letter or a number from a displayed list.

Our next program provides a general-purpose menu selection function. The screen messages, their screen coordinates and the addresses of the functions to be invoked are stored in a structure which is passed by address to the menu selection function. The function undertakes any selection requested. By writing a function in this fashion, it is easy to add extra options to a program with only minimal changes to existing code.

Program 23: menu

```
/***********************
*      MENU EXAMPLE      *
*-----------------------*
*     M.R.H.    May 85    *
***********************/

#include <stdio.h>

#define TRUE     1
#define FALSE    0
#define CLS     '\f'
#define stdin    0

struct menu_tmp
   {
   char x_pos;
   char y_pos;
   char *text_ptr;
   int (*funct)();
   };

char company[]= "GAMMA SYSTEMS LTD";

main()
   {
   main_process();
   rawout(CLS);
   }

dummy_process()
   {
   }

a5x_process()
   {
   not_available("A5 X");
   }

a5y_process()
   {
   not_available("A5 Y");
   }

a5z_process()
   {
   not_available("A5 Z");
   }

a1_process()
   {
   not_available("A1");
   }

a2_process()
   {
   not_available("A2");
   }
a3_process()
   {
   not_available("A3");
   }
```

```
a4_process()
    {
    not_available("A4");
    }

a5_process()
    {
    static struct menu_tmp a5_opts[] =
        {
        7,10," *** A5 ***",dummy_process,
        9,10,"OPTION A5 X ",a5x_process,
       10,10,"OPTION A5 Y ",a5y_process,
       11,10,"OPTION A5 Z ",a5z_process,
       13,10,"EXIT TO PREVIOUS MENU",dummy_process
        };

    menu_prompt(4,a5_opts);
    }

aaa_process()
    {
    static struct menu_tmp aaa_opts[] =
        {
        7,10," *** AAAAAAA ***",dummy_process,
        9,10,"OPTION A1",a1_process,
       10,10,"OPTION A2",a2_process,
       11,10,"OPTION A3",a3_process,
       12,10,"OPTION A4",a4_process,
       13,10,"OPTION A5",a5_process,
       15,10,"EXIT TO PREVIOUS MENU",dummy_process
        };

    menu_prompt(6,aaa_opts);
    }

bbb_process()
    {
    not_available("BBBBBBB");
    }

ccc_process()
    {
    not_available("CCCCCCC");
    }

ddd_process()
    {
    not_available("DDDDDDD");
    }
main_process()
    {
    static struct menu_tmp main_opts[] =
        {
        7,10," ***  MAIN MENU ***",dummy_process,
        9,10,"FACILITY AAAAAAA",aaa_process,
       10,10,"FACILITY BBBBBBB",bbb_process,
       11,10,"FACILITY CCCCCCC",ccc_process,
       12,10,"FACILITY DDDDDDD",ddd_process,
       14,10,"EXIT FROM SYSTEM",dummy_process
       };

    menu_prompt(5,main_opts);
    }
```

```
init_screen(msg_ptr)                                    /* MRH.LIB */
    char *msg_ptr;
    {
    int i;

    screen_mode(1);
    cursor_pos(1,10);
    printf("%s",msg_ptr);
    cursor_pos(2,1);
    for (i = 0; i < 40; i++)
       rawout('_');
    cursor_pos(22,1);
    for (i = 0; i < 40; i++)
       rawout('_');
    }

menu_prompt(no_options,ptr_options)                     /* MRH.LIB */
    int no_options;
    struct menu_tmp *ptr_options;
    {
    int j, val;
    char optn[4];

    init_scrn(company);
    while(TRUE)
       {
       for (j = 0; j <= no_options; j++)
          {
          cursor_pos((ptr_options + j)->x_pos,(ptr_options + j)->y_pos);
          if (j)
             printf("%2d %s", j,(ptr_options + j)->text_ptr);
          else
             printf("%s",(ptr_options + j)->text_ptr);
          }
       cursor_pos(23,1);
       printf("Select facility required: ");
       fgets(optn,4,stdin);
       val = atoi(optn);
       if (val < 1 || val > no_options)
          error_msg("Invalid option");
       else
          {
          if (val == no_options)
             return;
          else
             {
             (*(ptr_options + val)->funct)();
             init_scrn(company);
             }
          }
       }
    }

error_msg(msg_ptr)                                      /* MRH.LIB */
    char *msg_ptr;
    {
    cursor_pos(24,1);
    ink(3,3,13);
    paper(3);
    printf("%s",msg_ptr);
    paper(0);
    bell();
    };
```

```
not_available(ptr)                                  /* MRH.LIB */
   char *ptr;
   {
   init_scrn(company);
   cursor_pos(10,5);
   printf("Facility %s not available",ptr);
   cursor_pos(12,5);
   printf("Press any key to continue");
   while (!keyhit())
      ;
   rawin();
   }

#include ?stdio.lib?

#include ?mrh.lib?
```

A slight variation on this method is to move the cursor up and down using the cursor control keys until it is level with the option required. Then, by pressing a specified key, the required option is undertaken. This is done by keeping a counter which is decremented or incremented as the cursor is moved; on selection the counter corresponds to the option required.

9.9 Screen Handlers

In many programs it is common to find that the user is prompted line by line for information to be entered, For example:

```
Enter Input Device: $TERM
Enter Output Device: $LP
Enter Filename: OUTPAY
Enter Access Mode: READ ONLY
Enter Exclusion Mode: SHARED
```

The entry of such data would be greatly facilitated if all the screen prompts could be displayed simultaneously, and if the user could correct previously entered data. Such a routine that can handle this type of input is often referred to as a *Screen Requester* or a *Screen Handler*.

Program 24 illustrates another general purpose routine for handling Screen Requestors. As with the menu selection program, the definition of the screen is set up in a structure and is passed to the screen requestor function. The definition of the screen requires, for each data item to be input: a text string, screen coordinates, the address where the value is to be placed and the address of a function which validates the input item.

The user may tab between the data fields using the cursor left and right arrow keys. Cursor down will accept the screen and cursor up will abort. The Delete key will delete the last character while the CLR key deletes the complete field. The actual keys for these operations are defined at the top of the program and so could be changed if you feel that certain other keys are more convenient.

An example is given to input a name, address, etc. but the structure can easiliy be amended to handle any input data required. Numerical values must be entered as strings of characters and then converted.

Program 24: screen handler example

```
/*************************
* SCREEN HANDLER EXAMPLE *
*------------------------*
*     M.R.H.    May 85    *
*************************/

/*

#include <stdio.h>

*/

#define TRUE       1
#define FALSE      0
#define CLS      '\f'
#define CURSOR   207
#define SPACE     32
#define MARK      95
#define CURS_L     8
#define stdin      0

#define ACCEPT     241    /* CURSOR DOWN  KEY */
#define ABORT      240    /* CURSOR  UP   KEY */
#define TAB        243    /* CURSOR RIGHT KEY */
#define BTAB       242    /* CURSOR LEFT  KEY */
#define DELETE     127    /* DELETE KEY       */
#define CLR         16    /* CLEAR  KEY       */

struct screen_tmp
   {
   char x_pos;
   char y_pos;
   char length;
   char *text_ptr;
   int (*funct_ptr)();
   char *data_ptr;
   };

struct staff_tmp
   {
   char name[21];
   char adrss1[26];
   char adrss2[26];
   char adrss3[26];
   char adrss4[26];
   char phone[16];
   char dobirth[9];
```

```
    char comment[21];
    };

static struct staff_tmp staff_rec;

char company[]= "GAMMA SYSTEMS LTD";

int error_flg;

main()
    {
    while(TRUE)
        {
        if (get_staff() == ACCEPT)
            process_staff();
        else
            printf("%c\n\nABORTED",CLS);
        cursor_pos(24,1);
        printf("Do you want to continue ? ");
        if (!get_yn())
            return;
        }
    }

process_staff()
    {
    printf("%c\nENTERED DATA:",CLS);
    printf("\n\nNAME:     %s",staff_rec.name);
    printf("\n\nADDRESS: %s",staff_rec.adrss1);
    printf("\n          %s",staff_rec.adrss2);
    printf("\n          %s",staff_rec.adrss3);
    printf("\n          %s",staff_rec.adrss4);
    printf("\n\nPHONE:    %s",staff_rec.phone);
    printf("\n\nD O B:    %s",staff_rec.dobirth);
    printf("\n\nCOMMENT: %s",staff_rec.comment);
    }

dummy_process()
    {
    }

name_val()
    {
    if (staff_rec.name[0] == '\000')
        {
        cursor_pos(25,1);
        printf("MANDATORY FIELD");
        bell();
        error_flg = TRUE;
        return FALSE;
        }
    else
        return TRUE;
    }

no_val()
    {
    return TRUE;
    }

date_val()
    {
    int err;
```

```
    cursor_pos(24,1);
    err = validate_date(staff_rec.dobirth);
    if (!err)
       error_flg = TRUE;
    return (err);
    }

get_staff()
    {
    static struct screen_tmp staff_scrn[] =
       {
       6,  4, 20, "NAME:",     name_val,  staff_rec.name,
       8,  1, 25, "ADDRESS:", no_val,    staff_rec.adrss1,
       9,  9, 25, "",          no_val,    staff_rec.adrss2,
      10,  9, 25, "",          no_val,    staff_rec.adrss3,
      11,  9, 25, "",          no_val,    staff_rec.adrss4,
      13,  3, 15, "PHONE:",    no_val,    staff_rec.phone,
      15,  2, 8,  "D.O.B.:",   date_val,  staff_rec.dobirth,
      17,  1, 20, "COMMENT:", no_val,    staff_rec.comment
       };

    return (get_screen(8,staff_scrn));
    }
get_screen(no_fields,scrn_ptr)                    /* MRH.LIB */
    int no_fields;
    struct screen_tmp *scrn_ptr;
    {
    int v, j, k, idx, char_pos, valid;
    char key;

    init_scrn(company);
    error_flg = FALSE;
    for (j = 0; j < no_fields; j++)
       {
       cursor_pos((scrn_ptr + j)->x_pos,(scrn_ptr + j)->y_pos);
       printf("%s ",(scrn_ptr + j)->text_ptr);
       for (k = 0; k < (scrn_ptr + j)->length ; k++)
          {
          *((scrn_ptr + j)->data_ptr + k) = '\000';
          rawout(MARK);
          }
       }

    idx = 0;
    char_pos = 1;
    while (TRUE)
       {
       cursor_pos((scrn_ptr + idx)->x_pos, (scrn_ptr + idx)->y_pos +
                  strlen((scrn_ptr + idx)->text_ptr) + char_pos);
       rawout(CURSOR);
       key = rawin();
       if (error_flg)
          {
          error_flg = FALSE;
          clear_line(25,40);
          }
       cursor_pos((scrn_ptr + idx)->x_pos, (scrn_ptr + idx)->y_pos +
                  strlen((scrn_ptr + idx)->text_ptr) + char_pos);

       switch (key)
          {
          case ACCEPT:
             v = *((scrn_ptr + idx)->data_ptr + char_pos - 1);
```

```
            rawout((char_pos > (scrn_ptr + idx)->length) ? SPACE
                        : (v) ? v : MARK);
            valid = TRUE;
            for (idx = 0; idx < no_fields && valid; idx++)
               valid = (*(scrn_ptr + idx)->funct_ptr)();
            if (valid)
               return key;
            idx--;
            char_pos = 1;
            break;
case ABORT:
   return key;

case BTAB:
   if (idx > 0)
      {
      v = *((scrn_ptr + idx)->data_ptr + char_pos - 1);
      rawout((char_pos > (scrn_ptr + idx)->length) ? SPACE
                  : (v) ? v : MARK);
      idx--;
      char_pos = 1;
      }
   else
      bell();
   break;

case TAB:
   if (idx < (no_fields - 1))
      {
      v = *((scrn_ptr + idx)->data_ptr + char_pos - 1);
      rawout((char_pos > (scrn_ptr + idx)->length) ? SPACE
                  : (v) ? v : MARK);
      idx++;
      char_pos = 1;
      }
   else
      bell();
   break;

case CLR:
   cursor_pos((scrn_ptr + idx)->x_pos, (scrn_ptr + idx)->y_pos +
            strlen((scrn_ptr + idx)->text_ptr) + 1);
   for (k = 0; k < (scrn_ptr + idx)->length ; k++)
      {
      *((scrn_ptr + idx)->data_ptr + k) = '\000';
      rawout(MARK);
      }
   rawout(SPACE);
   char_pos = 1;
   break;

case DELETE:
   if (char_pos > 1 )
      {
      rawout((char_pos) > (scrn_ptr + idx)->length
                        ? SPACE : MARK);
      rawout(CURS_L);
      rawout(CURS_L);
      char_pos--;
      }
   else
      bell();
   break;
```

```
            default:
               if (char_pos <= (scrn_ptr + idx)->length && isprint(key))
                  {
                  rawout(key);
                  if (char_pos == 1)
                     {
                     cursor_pos((scrn_ptr + idx)->x_pos, (scrn_ptr + idx)
                    ->y_pos +strlen((scrn_ptr + idx)->text_ptr) + 2);
                     for (k = 1; k < (scrn_ptr + idx)->length ; k++)
                        {
                        *((scrn_ptr + idx)->data_ptr + k) = '\000';
                        rawout(MARK);
                        }
                     rawout(SPACE);
                     }
                  *((scrn_ptr + idx)->data_ptr + char_pos - 1) = key;
                  char_pos++;
                  }
               else
                  bell();
            }
         }

      }

clear_line(row,row_length)                          /* MRH.LIB */
   int row, row_length;
    {

    cursor_pos(row,1);
    while (row_length--)
       rawout(SPACE);
    }

/*

#include ?mrh.lib?

#include ?stdio.lib?

*/
```

9.10 Report Generation

In C when we talk to a computers I/O devices we do so by means of files. To communicate with the printer we open the list device just as any other file, and then write to it using functions such as `fprintf`. With the Amstrad life is simple and we can talk to the printer simply by using a file pointer with a value of 8.

example:

```
fprintf(8,"\nSIGMA PRESS");
```

Many modern day printers are very sophisticated and by sending certain control characters we can print in a number of styles and widths, however using them

in a program can be quite tedious. To make the production of elaborate reports simple, program 25 provides a general purpose routine that will enable a report to be specified, once again, by defining it in a structure. The items that must be specified are: its column position, the data length and the address of the data item. In addition to data items, control characters may be specified by stating a item length of CONT_CHAR (this is defined at the top of the program).

Program 25 uses the screen requestor routine from the previous program to accept certain items of data for the production of a Staff Pay and Expenses Slip.

Program 25: staff pay & expenses

```
/*************************
*   REPORT GENERATION    *
*-----------------------*
*     M.R.H.   May 85    *
*************************/

#include <stdio.h>

FILE *printer_stream = 8;    /* 8 = AMSTRAD  */
                             /* 3 = SPECTRUM */

/* The following control characters may require amendments to
   suit your own printer.  Since Null characters are interpreted
   in C as the end of a string, they must be represented by the
   character @
*/

#define S_UNDRLN       "\033-\1" /* set underline on     */
#define R_UNDRLN       "\033-@"  /* reset underline      */
#define S_ITALIC       "\0334"   /* set italic on        */
#define R_ITALIC       "\0335"   /* reset italic         */
#define S_EMPHZE       "\033E"   /* set emphasize on     */
#define R_EMPHZE       "\033F"   /* reset emphasize      */
#define S_D_STRK       "\033G"   /* set double strike    */
#define R_D_STRK       "\033H"   /* reset double strike */
#define S_D_WIDH       "\016"    /* set double width on */
#define R_D_WIDH       "\015"    /* reset double width  */
#define LINEFEED       "\012"    /* linefeed             */
#define FORMFEED       "\014"    /* formfeed             */

#define TRUE            1
#define FALSE           0
#define CLS           '\f'
#define CURSOR        207
#define SPACE          32
#define MARK           95
#define CURS_L          8
#define stdin           0
#define CONT_CHAR  '\177'

#define ACCEPT      241    /* CURSOR DOWN  KEY */
#define ABORT       240    /* CURSOR  UP   KEY */
#define TAB         243    /* CURSOR RIGHT KEY */
#define BTAB        242    /* CURSOR LEFT  KEY */
#define DELETE      127    /* DELETE KEY       */
#define CLR          16    /* CLEAR  KEY       */
```

```
struct screen_tmp
   {
   char x_pos;
   char y_pos;
   char length;
   char *text_ptr;
   int (*funct_ptr)();
   char *data_ptr;
   };

struct report_tmp
   {
   char info_pos;
   char info_len;
   char *info_ptr;
   };

int error_flg, char_cnt;

char name[21], period[16], tax_code[6], gross_pay[9],
     tax[9], nat_ins[9], expenses[9], total[9], comment[21];

char company[] = "GAMMA SYSTEMS LTD";

main()
   {
   while(TRUE)
      {
      if (get_data() == ACCEPT)
         process_data();
      else
         printf("%c\n\nABORTED",CLS);
      cursor_pos(24,1);
      printf("Do you want to continue ? ");
      if (!get_yn())
         return;
      }
   }

no_val()
   {
   return TRUE;
   }

get_data()
   {
   static struct screen_tmp data_scrn[] =
      {
       6,  7, 20, "NAME:",       no_val,   name,
       8,  5, 15, "PERIOD:",     no_val,   period,
      10,  4,  4, "TAXCODE:",    no_val,   tax_code,
      12,  2,  8, "GROSS PAY:", no_val,   gross_pay,
      13,  8,  8, "TAX:",        no_val,   tax,
      14,  4,  8, "NAT INS:",    no_val,   nat_ins,
      15,  3,  8, "EXPENSES:",   no_val,   expenses,
      16,  6,  8, "TOTAL:",      no_val,   total,
      18,  4, 20, "COMMENT:",    no_val,   comment,
       };

   return (get_screen(9,data_scrn));
   }
```

```
process_data()
   {
   static struct report_tmp pay_format[] =
      {
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   S_EMPHZE,
      0,  CONT_CHAR,   S_D_WIDH,
      0,         20,   company,
      0,  CONT_CHAR,   R_D_WIDH,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   S_UNDRLN,
      1,         22,   "Pay and Expense Advice",
      0,  CONT_CHAR,   R_UNDRLN,
      0,  CONT_CHAR,   R_EMPHZE,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   LINEFEED,
      1,          5,   "NAME:",
      9,         20,   name,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   LINEFEED,
      1,          7,   "PERIOD:",
      9,         15,   period,
     30,          9,   "TAX CODE:",
     40,          4,   tax_code,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   LINEFEED,
      1,         50,   "GROSS                    NATIONAL",
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   S_UNDRLN,
      1,         56,   "PAY         TAX    INSURANCE      EXPENSES      TOTAL
      0,  CONT_CHAR,   R_UNDRLN,
      0,  CONT_CHAR,   LINEFEED,
      1,          8,   gross_pay,
     12,          8,   tax,
     22,          8,   nat_ins,
     36,          8,   expenses,
     49,          8,   total,
      0,  CONT_CHAR,   LINEFEED,
      0,  CONT_CHAR,   LINEFEED,
      1,         20,   comment,
      0,  CONT_CHAR,   FORMFEED
     };

   cursor_pos(24,1);
   printf("Do you want to print payslip ? ");
   if (get_yn())
      print_lines(38,pay_format);
   clear_line(24,40);
   }

print_lines(item_no,item_ptr)                    /* MRH.LIB */
   int item_no;
   struct report_tmp *item_ptr;
   {
   int j, pos, len;

   char_cnt = 0;
   for (j = 0; j < item_no; j++)
      {
      pos = (item_ptr + j)->info_pos;
      len = (item_ptr + j)->info_len;
      if (len != CONT_CHAR)
```

```
            {
            for ( ; char_cnt < pos; char_cnt++)
               putc(SPACE,printer_stream);
            char_cnt += print_data(len,(item_ptr + j)->info_ptr);
            }
         else
            {
            print_data(CONT_CHAR,(item_ptr + j)->info_ptr);
            if (strcmp((item_ptr + j)->info_ptr,LINEFEED) == 0)
               char_cnt = 0;
            }
         }
      }

print_data(data_len,data_ptr)              /* MRH.LIB */
   int data_len;
   char *data_ptr;
   {
   int chars_inp;

   for (chars_inp = 0; chars_inp < data_len; chars_inp++)
      {
      switch (*(data_ptr + chars_inp))
         {
         case NULL:
            return chars_inp;
         case '@':
            putc((data_len == CONT_CHAR) ? '\000' : '@',printer_stream);
            break;
         default:
            putc(*(data_ptr + chars_inp),printer_stream);
            break;
         }
      }
   return chars_inp;
   }

#include ?mrh.lib?

#include ?stdio.lib?
```

Output from program:

```
GAMMA SYSTEMS LTD

Pay and Expense Advice

NAME:    STEPHANIE J CLOUGH

PERIOD: December 1985          TAX CODE: 200L

GROSS                    NATIONAL
PAY          TAX         INSURANCE      EXPENSES      TOTAL
875.00       212.10      78.84          46.85         630.91

Happy New Year
```

9.11 Time Input / Firmware System Calls.

Most micros have an internal counter which comes into effect the moment the machine is switched on. The counter is incremented at a steady rate, typically at 1/300th of a second, and is extremely accurate. By using this facility we can add timing to our programs, for example as a clock, stop-watch or for timing moves in games. We could also time the lengths of processes to compare the efficiency of different programming algorithms. To see how to access this facility on your system you should consult the documentation provided with your machine.

The Amstrad contains a four byte counter which is incremented every 1/300th of a second and can be accessed using the firmware routines. This can be done neatly using the Hisoft `inline` function which provides a mechanism for linking machine code into your C. Two calls can be made: one to read the clock and another to reset it to a particular value. For further information on these and other firmware calls, you are advised to read the Amstrad firmware manual.

The next two programs illustrate calling the firmware timing routines.

The first, program 26, obtains the lower sixteen bits of the clock and uses it to seed the random number generator to randomize the position in the list of sequential numbers. The program is a game of skill (a numeric form of Mastermind) in which you have to crack a four digit code consisting of numbers in the range of 1 to 6.

You enter your attempt and study the clues that are returned. The clue consists of two digits. The first gives the number of attempt digits that are correct and in the correct position; the second gives the number of digits that are correct but in the wrong position. The game stops when the correct code has been entered.

Example:

Attempt	*Clue*
4321	2/0
6565	0/1
5221	2/0
6621	1/0
3531	1/2
3511	2/2
5311	4/0

Program 26: code breaking

```
/*************************
*      CODE BREAKER       *
*-----------------------*
*      MRH  May 85        *
*************************/

#include <stdio.h>

#define TRUE          1
#define FALSE         0

#define CALL                  0xCD
#define LD_HL_into            0x22
#define KL_TIME_PLEASE        0xBD0D

int a[4], b[4], c[4];
int j, k, dot, star;

main()
   {
   printf("CODE BREAKER");
   randomize();
   while (TRUE)
      {
      process();
      printf("\nPlay again ? ");
      if (!get_yn())
         return;
      }
   }

process()
   {
   for (j = 0; j < 4; j++)
      b[j] = rnd(6);
   printf("\n\n");
   while (!get_attempt())
      ;
   }
```

```
get_attempt()
    {
    dot = 0; star = 0;
    printf("\nEnter attempt ");
    for (j = 0; j < 4; j++)
        {
        c[j] = get_digit(1,6);
        a[j] = b[j];
        }
    for (j = 0; j < 4; j++)
        if (a[j] == c[j])
            {
            star++;
            a[j] = -1;
            c[j] = -2;
            }
    for (j = 0; j < 4; j++)
        for (k = 0; k< 4; k++)
            if (a[j] == c[k])
                {
                dot++;
                a[j] = -1;
                c[k] = -2;
                }
    printf("  %d/%d ",star,dot);
    if (star == 4)
        {
        printf("WELL DONE");
        return;
        }
    }

get_digit(min,max)          /* MRH.LIB */
    int min,max;
    {
    int v;

    while (TRUE)
        {
        while (!keyhit())
        ;
```

```
        v = rawin() - 48;
        if (v >= min && v <= max)
            {
            printf("%d",v);
            return v;
            }
        }
    }

randomize()                    /* MRH.LIB */
    {
    static int seed;

    inline(CALL, KL_TIME_PLEASE,
           LD_HL_into, &seed);
    srand(seed);
    }

#include ?mrh.lib?

#include ?stdio.lib?
```

Program 27 illustrates the clock facility to time the speed of your reactions. Two functions are used, one to reset the clock and the other to read the elapsed time.

Program 27:

```
/************************
*    REACTION TIMER     *
*-----------------------*
*     MRH  May 85       *
************************/

#include <stdio.h>

#define TRUE         1
#define FALSE        0
#define SPACE       32
#define CLS        '\f'

#define CALL                0xCD
#define LD_HL_into          0x22
#define LD_DE_into          0xED, 0x53
#define LD_HL_with          0x2A
#define LD_DE_with          0xED, 0x5B
#define TIME_PLEASE         0xBD0D
#define TIME_SET            0xBD10

int time_array[2];
```

```
main()
   {
   screen_mode(1);
   printf("REACTION TIMER");
   while (TRUE)
      {
      printf("\n\nPress [SPACE BAR] to play");
      while (!keyhit())
         ;
      if (rawin() != SPACE)
         return;
      rawout(CLS);
      time_array[1] = time_array[0] = 0;
      while (rnd(100) != 1)
         ;
      cursor_pos(6,10);
      printf("BANG!!");
      bell();
      set_time(time_array);
      while (!keyhit())
         ;
      get_time(time_array);
      rawin();
      if (time_array[0])
         printf("\n\n\nReacted in %d hundredths of a second",
                                 time_array[0] / 3);
      else
          printf("\n\n\nI am sorry but you cheated");
      }
   }

get_time(time_ptr)                  /* MRH.LIB */
   int *time_ptr;
   {
   static int reg_de, reg_hl;

   inline(CALL, TIME_PLEASE,
          LD_HL_into, &reg_hl,
          LD_DE_into, &reg_de);
   *time_ptr = reg_hl;
   *(time_ptr + 1) = reg_de;
   }

set_time(time_ptr)                  /* MRH.LIB */
   int *time_ptr;
   {
   static int reg_de, reg_hl;

   reg_hl = *time_ptr;
   reg_de = *(time_ptr + 1);
   inline(LD_HL_with, &reg_hl,
          LD_DE_with, &reg_de,
          CALL, TIME_SET);
   }


#include ?mrh.lib?

#include ?stdio.lib?
```

9.12 High Resolution Graphics

Up to now, we have always considered the display screen to be divided up into a number of character positions. However, most screens can be divided up further into a grid of smaller pixel dots that can either be illuminated or not. The Amstrad, which is fairly typical, consists of 640 by 200 pixels in screen mode 2. All systems will have different methods for controlling these pixels and so, once again, you will have to refer to the documentation provided with your compiler to find the appropriate methods.

To control the pixels on the Amstrad we shall use the `inline` statement to call routines in the Amstrad firmware. In program 28, in addition to a touch of modern art, are three useful functions for plotting a point, drawing a line from the current plot position and for setting up the graphics pen. For detailed information on them you should consult the Amstrad firmware manual.

Program 28: Picasso

```
/*************************
*       PICCASSO          *
*-----------------------*
*      MRH  May 85        *
*************************/

#define LD_A_with           0x3A
#define LD_HL_with          0x2A
#define LD_DE_with          0xED,0x5B
#define CALL                0xCD
#define GRA_PLOT_A          0xBBEA
#define GRA_DRAW_A          0xBBF6
#define GRA_SET_PEN         0xBBDE

#define CLS                 '\f'

main()
    {
    ink(0,24,24);
    ink(2,6,6);
    screen_mode(1);
    rawout(CLS);
    process_screen();
    rawin();
```

```
    rawout(CLS);
    screen_mode(2);
    ink(0,1,1);
    }

process_screen()
    {
    int x;

    for (x = 0; x < 640; x++)
        {
        if (keyhit())
            return;
        set_graphics_pen(x % 4);
        plot(x, 0);
        draw(640 - x, 399);
        }
    for (x = 0; x < 400; x++)
        {
        if (keyhit())
            return;
        set_graphics_pen(x % 4);
        plot(0, x);
        draw(639, 400 - x);
        }
    }


plot(x,y)                          /* MRH.LIB */
    int x,y;
    {
    static int reg_de, reg_hl;

    reg_hl = y;
    reg_de = x;
    inline(LD_HL_with, &reg_hl,
           LD_DE_with, &reg_de,
           CALL, GRA_PLOT_A );
    }
```

```
draw(x,y)                              /* MRH.LIB */
    int x,y;
    {
    static int reg_de, reg_hl;

    reg_hl = y;
    reg_de = x;
    inline(LD_HL_with, &reg_hl,
           LD_DE_with, &reg_de,
           CALL, GRA_DRAW_A );
    }


set_graphics_pen(ink)                  /* MRH.LIB */
    char ink;
    {
    static char reg_a;

    reg_a = ink;
    inline(LD_A_with,&reg_a,
           CALL, GRA_SET_PEN);
    }

#include ?mrh.lib?
```

Program 29 will plot a three dimensional histogram for up to 18 values. The main problem when drawing graphs is to choose a suitable scale for the data. By inspecting the program you will see that a block of 300 units is drawn for the largest item of data and that all the other blocks are drawn in proportion.

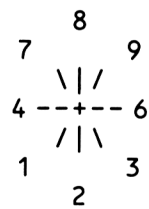
Program 29:

```
/************************
*    3D HISTOGRAM       *
*-----------------------*
*     MRH  May 85       *
************************/

#define TRUE         1
#define FALSE        0
#define CLS        '\f'

int n, data[18];

 main()
   {
   while (TRUE)
      {
```

```
        if (enter_data())
            {
            plot_histogram();
            printf("\nDo you want to continue ? ");
            if (!get_yn())
               return;
            }
        }
    }

enter_data()
    {
    int i;

    printf("\n\nEnter number of items (5 - 18) : ");
    scanf(" %d",&n);
    if (n < 5 || n > 18)
        {
        bell();
        printf("\nInvalid value");
        return FALSE;
        }
    printf("Enter data (between 0 and 2000):\n");
    for (i = 0; i < n; i++)
        {
        scanf(" %d",&data[i]);
        if (data[i] > 2000)
            {
            bell();
            printf("\nInvalid value");
            return FALSE;
            }
        }
    return TRUE;
    }

plot_histogram()
    {
    int i;
    unsigned len, max;

    screen_mode(1);
    pen(2);
    printf("THREE DIMENSIONAL PLOT");
    max = 0;
    for (i = 0; i < n; i++)
        {
        if (data[i] > max)
            max = data[i];
        }
    for (i = 0; i < n; i++)
        {
        len = 30 * data[i] / max * 10;    /* must not overflow 16 bits */
        draw_bar(i, len);
        }
    if (!keyhit())
        ;
    rawin();
    pen(1);
    screen_mode(2);
    ink(1,24,24);
    }
```

```
draw_bar(pos,len)
    unsigned pos, len;
    {
    int i;

    ink(2,8,8);
    ink(3,7,7);
    ink(1,4,4);
    for (i = 0; i < len; i++)
        {
        plot(32 * pos,i);
        set_graphics_ink(2);
        draw(32 * pos + 30, i);
        set_graphics_ink(3);
        draw(32 * pos + 40, i + 10);
        }
    set_graphics_ink(1);
    for (i = 0; i < 10; i++)
        {
        plot(32 * pos + i, len + i);
        draw(32 * pos + i + 32, len + i);
        }
    }

#include ?mrh.lib?
```

In our final example, program 30, we can draw pictures by plotting lines and curves under the control of the user. The direction of a plotted line is chosen by selecting one of the following keys:



By pressing key B, the colour will be switched to the background colour; this is useful if part of the picture is to be erased or if the current plot position is to be moved elsewhere without leaving a trailing line.

By pressing N, the colour will return to normal foreground colour.

Finally, by pressing C the screen is cleared.

Program 30:

```
/************************
*    ETCHA SKETCH       *
*-----------------------*
*     MRH  May 85       *
************************/

#define TRUE           1
#define FALSE          0
#define CLS          '\f'

int key, x_pos, y_pos;

main()
   {
   x_pos = 320;
   y_pos = 200;
   screen_mode(1);
   process();
   }

process()
   {
   rawout(CLS);
   while (TRUE)
      {
      key = rawin();
      if (x_pos > 0   && (key == '7' || key == '4' || key == '1'))
         x_pos -= 2;
      if (x_pos < 638 && (key == '9' || key == '6' || key == '3'))
         x_pos += 2;
      if (y_pos > 0   && (key == '1' || key == '2' || key == '3'))
         y_pos -= 2;
      if (y_pos < 398 && (key == '7' || key == '8' || key == '9'))
         y_pos += 2;
      if (key == 'C' || key == 'c')
         rawout(CLS);
      if (key == 'X' || key == 'x')
         return;
      if (key == 'N' || key == 'n')
         set_graphics_pen(1);
      if (key == 'B' || key == 'b')
         set_graphics_pen(0);
      plot(x_pos,y_pos);
      }
   }

#include ?mrh.lib?
```

APPENDIX A.

THE STANDARD C LIBRARY.

APPENDIX B.

THE MRH C LIBRARY.

Function **Page**

APPENDIX C.

THE HISOFT C LIBRARY.

The Hisoft C library is supplied in several files:

"stdio.h" contains commonly used constants and forward declarations for library functions

"stdio.lib" contains machine independent library functions based on those supplied with Unix systems (see Appendix A)

"firmware.lib" contains simple functions to access the firmware jumpblock

"basic.lib" contains functions to provide access to the computers facilities in a way that is intended to be familiar to the BASIC programmer.

The Hisoft BASIC library includes the following functions:

```
after
border
cass_speed
catalog
clear_graphic_screen
draw
drawr
event_disable
event_enable
flash_speed
ink
inkey
inp
instr
itob
joy
```

```
key_function
key_speed
key_translation
out
plot
plotr
sound
sound_check
strlower
strupper
symbol
symbol_after
time
write_file
```

APPENDIX D.
FURTHER READING

"The Amstrad Advanced User Guide" – *Mark Harrison, Sigma Press.*

"The Sinclair Spectrum in Focus" – *Mark Harrison, Sigma Press.*

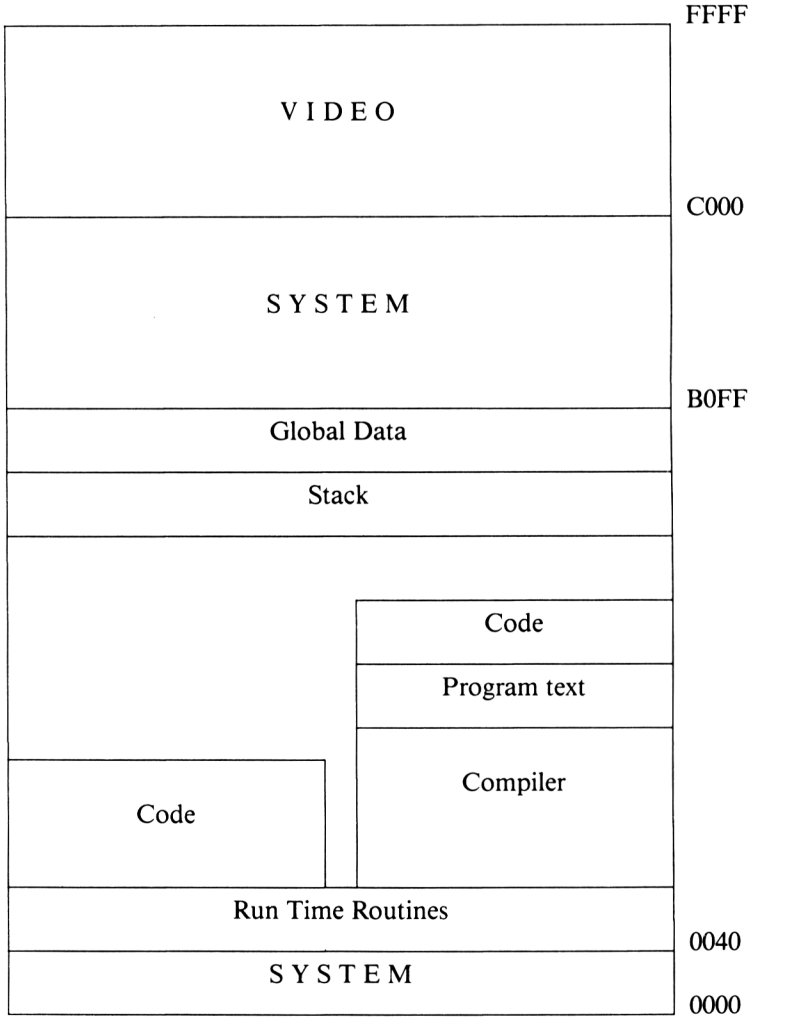
"Successful Software for Small Computers" – *Graham Beech, Sigma Press.*

"The Big Red Book of C" – *Kevin Sullivan, Sigma Press.*

"The Amstrad Firmware Manual" – *Amsoft*

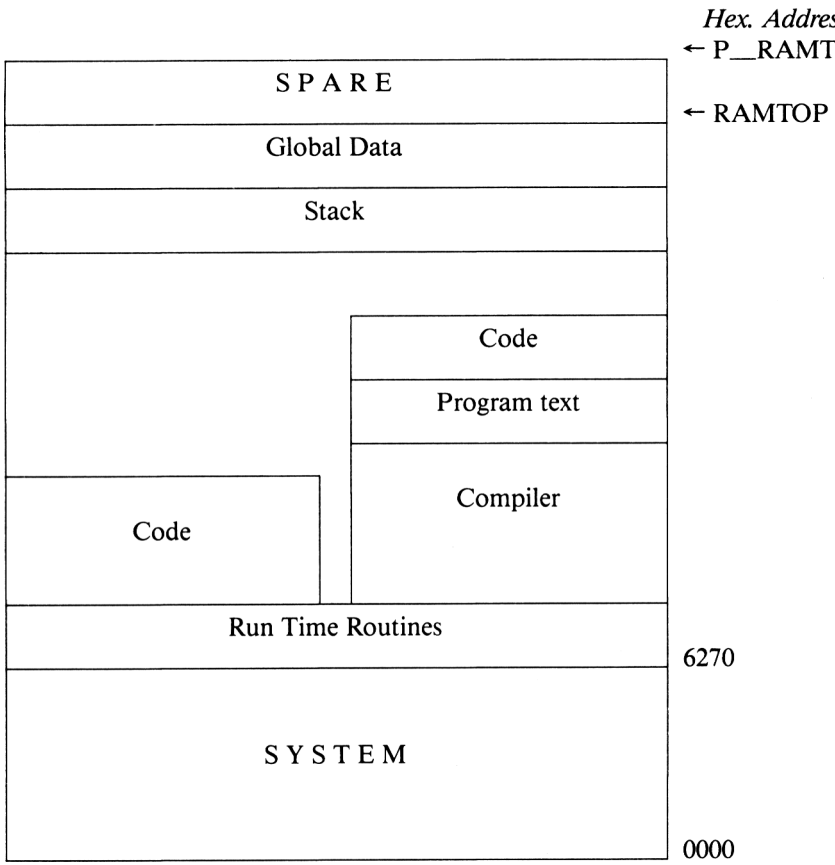
APPENDIX E.

The Amstrad/Hisoft Memory Map



APPENDIX F.

The Spectrum/Hisoft Memory Map



INDEX

C commands and statements beginning with a non-alphabetic symbol:

In the remainder of this index, entries beginning with a lower case letter are C commands or statements.

How to use this book – and avoid C sickness!!

'Practical C' is a collection of ideas and techniques written for both the newcomer to the C programming language and the experienced C programmer who wishes to get the most from an implementation of the language on a personal computer. An elementary knowledge of programming is assumed.

The book starts with the basic principles of the language and then, building on this introduction, the reader is taken on a programming course which leads up to some of the most advanced techniques that can be used with the language. Some readers may prefer to use the earlier sections of the book as a reference manual and jump into the sections devoted to programming techniques and general purpose routines.

All of the techniques which are introduced are illustrated by tried and tested programs written using the Hisoft implementation of the language on an Amstrad CPC 464/6128. However, C programs can be easily ported on to other machines and so all the programs will work, with just minimal changes, using other versions of the language. Any differences between Hisoft C and the true definition of the language are highlighted for the benefit of readers using other compilers.

Many of the C routines listed will provide the reader with advanced C tools. By working through the book, the reader can build up a library of functions which will become invaluable in future program development. A list of functions used in the book, along with the Standard C and Hisoft C functions is given for reference in the Appendices.

About the Author

Mark Harrison is a freelance systems consultant providing consultancy and microcomputer systems to business and comerce. His group specialises in applying multi-user microcomputers to various applications and concentrates on systems implemented using the C programming language and operating under the Unix or CP/M operating system.

Mark Harrison (Systems and Technology) Ltd., may be contacted at: 12A Merton Close, Owlsmoor, Camberley, Surrey GU15 4TU.

GB £ NET +007.95

ISBN 1-85058-035-9

About Us

Sigma Press specialise in computing, science, & technology. Write for a catalogue or to tell us about your own ideas for a book:

Sigma Press
98a Water Lane
Wilmslow
Cheshire SK9 5DT

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/