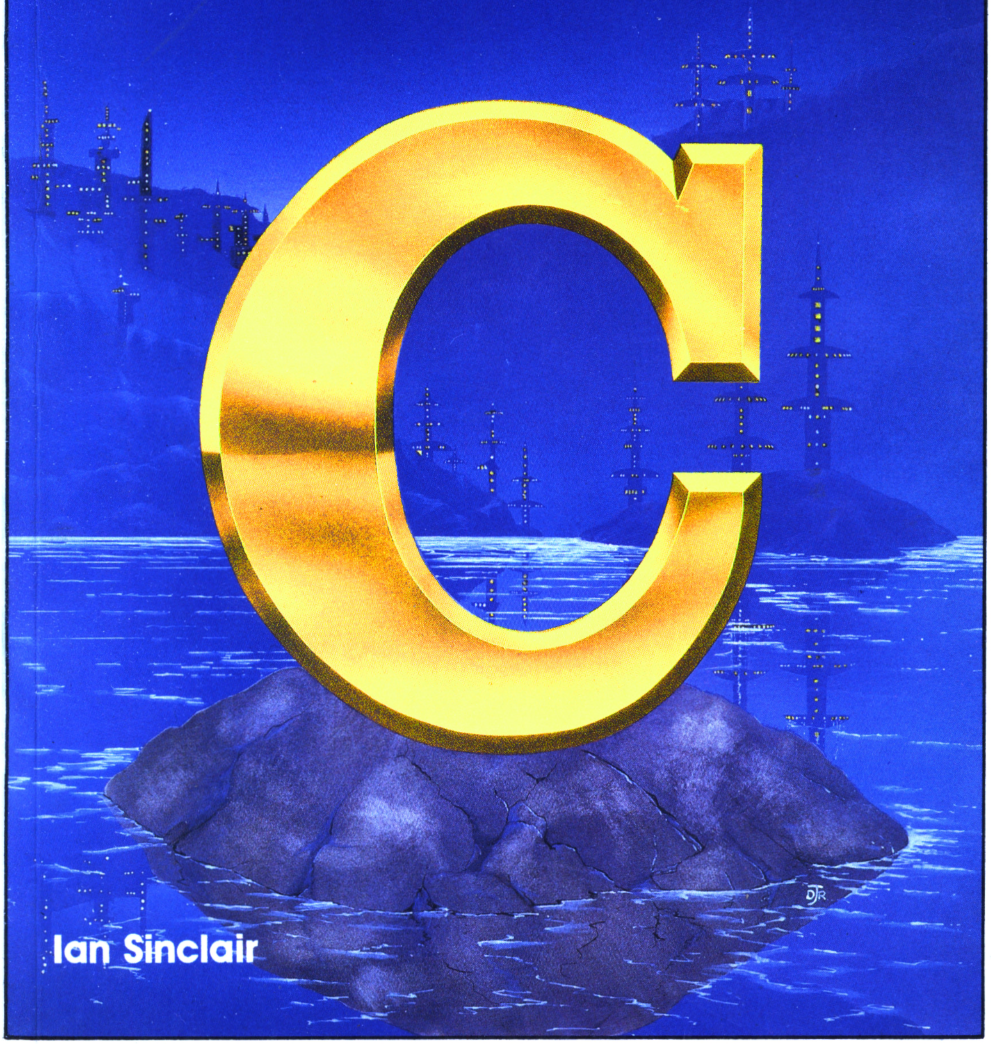
C FOR BEGINNERS



Ian Sinclair

C FOR BEGINNERS

Ian Sinclair

MELBOURNE HOUSE
PUBLISHERS

Published in United Kingdom by
Melbourne House (Publishers) Ltd
60 High Street, Hampton Wick
Kingston-Upon-Thames
Surrey KT1 4DB

Published in Australia by
Melbourne House (Australia) Pty Ltd
96-100 Tope Street
South Melbourne, Victoria 3205

ISBN 0 86161 206 X

First Printing 1986

Printed and bound in Great Britain by Short Run Press Ltd., Exeter

Contents

Preface

Writing a book of a general nature is always fraught with difficulties, and particularly when the book concerns a language. I don't know, for example, which computer you may be using or which C compiler you have running, or if you have used any programming language previously, so that everything here is always subject to the unwritten proviso 'if your machine/compiler supports this...'. Fortunately, the C language itself is well standardised, so that the fundamentals that appear here will apply to any suitable combination of machine and compiler, though the examples have obviously been produced with one specific machine, the Amstrad, and compiler (Hisoft C) combination. Some care has been taken to ensure that most of these examples should run on whatever machine you happen to use. They have been printed with forty characters per line in order to make reproduction simpler and to correspond with the appearance of the listing on a 40-character screen.

This is a book intended for the true beginner to C. There are many books on the C language, all of which to some extent owe their material to the source book, *The C Programming Language,* by Kernighan and Ritchie. Kernighan & Ritchie's book is a definitive statement of the C language, and anything that differs from their version is not standard C. Their book is not in any sense a beginner's guide, however, and this text is an attempt to introduce you in a gentler way to this very significant and useful language. This means that the language is not exhaustively described in every possible detail, nor is every possible combination of statements explored.

What I have attempted to do is to introduce C to the reader who has probably programmed to some extent in BASIC, and to show the similarities and the very considerable differences between these languages. I hope that in the course of doing so I have illustrated how flexible, useful and beautiful the C language is. Once you have worked your way through this book and, more importantly, worked with the examples on your own computer, then if you are serious about C you should buy a copy of Kernighan & Ritchie's book.

All of the examples have been prepared with the Hisoft-C compiler running on the Amstrad CPC6128, and I am most grateful to Dave Howarth of Hisoft for many discussions and much illumination. I am most grateful also to my friends at Melbourne House, particularly Alan Giles, Rachel Davies, and Geoff Halprin, who have made this book a reality.

Ian Sinclair, Spring 1986.

Chapter 1
What is 'C'?

'C' is the curiously short name for one of the great computing languages of our time. The name is an historical accident, as I'll show you later, but for now we'll concentrate on the what and why of it.

To start with, we have to look at programming languages generally. At the bottom of the heap of programming level comes machine code, writing directly in binary or hex digits. Machine-code programming is tedious, error-prone and liable to hold a bug in every byte. There are three main reasons for using it. One is that it's the only way to program a completely new microprocessor for which nothing is written. Another is its speed, and the third is to achieve machine-control. The first reason applies only if you write language compilers for new microprocessors, but the other two reasons are the ones that keep machine code programmers in business. Machine code is fast - there's nothing faster, and for some types of programming, notably animated graphics, this speed is essential. The other reason is also important. There are always features of a machine that can be controlled only by machine code. No language like BASIC can ever cope with every possibility. If you want to have your screen scrolling sideways, or to run an unusual disc system, or to use a non-standard printer then you need machine code to write the routines.

There is such a continuing need for machine code that we need a programming language, assembly language, for each microprocessor type in order to write machine code with less tedium and fewer bugs. Assembly language, then, comes slightly higher up the scale compared to machine code, because it's easier to write, but it still translates directly to machine code.

What about the other end of the scale? There's one 'high-level' language, COBOL, which looks almost like a set of instructions in English when you read a program written in this language. When you write programs in a language of this type, you don't expect to have to worry about the details of the machine. You aren't interested in what processor unit it uses (on a mainframe, you don't have a microprocessor, just a central processor unit) or how many bits make a byte. You don't need to know ASCII codes, or routine addresses in ROM, or any of the things that constantly occupy the minds of assembly language programmers. You simply write your program lines, run them into the machine, and sit back like anyone else until the program crashes. The language processor (a compiler) converts the instructions of your program into machine code, and the computer executes the machine code.

The name 'high-level' is a good one - you are so far above the ground level of machine code that you hardly know there's a machine there. Needless to say, the language is the same no matter which machine you happen to be programming.

Between the heights of COBOL (and also FORTRAN and ALGOL) and the ground level of machine code, there are languages at all sorts of intermediate levels.

The fundamental problem is that high level languages are powerful but inefficient. They allow you to turn your problem-solving methods into programs that run smoothly, but at a great cost in memory space. They can be cumbersome, using lines and lines of program which can take all day to compile and run. At the other end, machine code is very compact, very fast, very efficient - but sheer hell to write and debug in any quantity.

The reason that we have such a large number of programming languages is that we are constantly trying to get a better balance of these different virtues. What we want is a high level language that makes it easy to express our solutions, is very compact both in statement length and use of memory, and which translates into as few bytes of machine code as would be provided by an assembler. There's no such language, and probably never will be, but some languages come nearer to the ideal than others, and some are a better solution for some kinds of problems. For example, a lot of programming of the type used in robotics is well handled by the language FORTH, but you would want a truckload of aspirin if you intended to write a word-processor in FORTH.

The history of computing, then, is littered with languages that seemed a good idea at the time, or which looked like being a good solution to one type of need. The greatest computing languages are the ones which have incorporated ideas that have lasted and have passed into other languages. For example, one of the first great languages, FORTRAN, has been so incorporated into BASIC that a FORTRAN programmer can easily switch to BASIC without feeling that this is a different sort of language. The other great language of early computing is ALGOL - and that's where we start the story of 'C'.

The name of C is almost an accident, derived from the history of the development of the language. A remote ancestor of 'C' was a language called CPL, meaning Combined Programming Language. This was evolved around 1963 in the Universities of Cambridge and of London as an attempt to make a more practical version of ALGOL which had some of the features of the earlier FORTRAN.

ALGOL is a very different type of language from FORTRAN or BASIC, one of the first of the modern computer languages, and one which did not find much favour with programmers in its day. Looking back, it's not easy to see why ALGOL was so rejected, and modern versions are still regarded among the most effective computer languages for large machines. One reason that was given at the time was that ALGOL seemed too much of a general-purpose language, unlike FORTRAN (for scientific and engineering uses) or COBOL (for business applications).

Be that as it may, CPL fared little better in the popularity stakes because it ended up as a big language that needed a lot of memory for its compiler. It also had a large number of actions that required a lot of effort to learn and were not necessarily particularly useful to the commercial programmer.

The ideas that had passed from ALGOL to CPL, however, were too good to be allowed to escape, and Martin Richards at Cambridge came up with a new version in 1967 - BCPL, the 'B' meaning 'Basic'.

The aim of BCPL was to create a compact language that could be used with small memory capacities, but retaining the best features of CPL. BCPL is still around, and there is a BCPL compiler for the BBC Micro, but both BCPL, and a more simplified version 'B', suffer from being *too* brief.

When a language becomes too brief, like some of the early 4K BASICs, it becomes too limited for really effective programming, and you find yourself writing ten lines where one would suffice in a richer language.

The significance of 'B', however, is that it was written at Bell Laboratories, by Ken Thompson (1970) at a time when the Unix operating system was being developed for main-frame computers. Working also at Bell Labs was Dennis Ritchie, and in 1972, at the start of the microprocessor age, he invented 'C', one year before Gary Kildall devised the CP/M operating system. The name was simply a follow-on to 'B', and the aim was to make the language more generally useful, particularly for writing operating systems. Unix was still under development, and a language that could make it easier to develop routines was to prove very useful.

'C' is a language which has a level midway between COBOL and machine code. It's an economical language, meaning that a few short statements can accomplish a lot of action. It's also a language that is very rich in control structures, so that you can form loops easily and escape from them equally easily, without the use of GOTOs. It has a good range of data types, so that you can work with any type of data, number, character, string, array and so on. Its main glory, however, is that it is remarkably free from the restrictions that a lot of high level languages place on the programmer.

Just to give you an example of what I mean by this, PASCAL allows you to define a string as being an array of a given number of characters. If you define NAME as being a string of ten characters, then you can't assign the name SINCLAIR to the variable NAME, because there are only 8 characters in SINCLAIR. To make the assignment work, you have to add two spaces to SINCLAIR to make the length up to ten characters. This is the type of restriction that is unknown in most varieties of BASIC, and which is also less of a problem in 'C'. This absence of restrictions makes 'C' a very useful language for programmers who are working with machine utilities and systems, but the lack of restriction is bought at a price.

'C' will let you do practically anything - and that includes making mistakes. Hand in hand with the lack of restrictions goes lack of checks - as long as your syntax looks reasonable, you can make any mistake you like, and you don't know any more about it until the program crashes. Even seasoned 'C' programmers can put very elusive bugs into programs - in fact the more experienced you get, the more elusive your bugs become. Given the choice, though, most programmers will accept the risks in order to write efficient programs. Nothing's perfect. You might not cut yourself on a blunt chisel, but it's not much use on wood. Given the choice, craftsmen choose sharp chisels and learn not to cut themselves. In computing language terms, 'C' is a very sharp chisel.

Compilers and interpreters

Like many other languages, 'C" can be interpreted or compiled Whatever language you use to express your program in, it has to be converted into machine code before it can have any effect on the computer. Interpreting and compiling are two methods of carrying out this conversion. When a language is interpreted, each instruction is taken in turn as the interpreter comes to it. It is checked to see if the instruction is valid, and then executed by calling up the machine-code subroutine before carrying on to the next instruction. In practical terms, this means that a set of machine code subroutines exist on disc or in ROM, and each reserved word of the language calls up a routine or set of routines. In a loop, each instruction in the loop will be interpreted, checked, and run on each pass through the loop, and this is what makes an interpreter slow in action.

A compiled language, by contrast, converts all of the instructions of a program into a large machine code program, by the same kind of action as interpretation with no error checking. The result is very often recorded on disc rather than being run at the time, and the action of translating from high level language into machine code is called compiling. Once the program has been compiled, it is a machine code program, just as it would have been if created using an assembler. If the compiler is a good one, the machine code may be almost as compact as it would have been if created by an assembler (referred to as 'native' machine code).

Nothing's perfect, and both interpreters and compilers have merits and faults. The main merit of an interpreter is easy debugging. When your interpreted program stops with an error message, you can change the text and start again. For a compiled program, this would mean loading in the text, changing part of it, compiling and recording the machine code, and then running again.

The disadvantage of an interpreter is low speed, because the process of finding the correct machine code routines for each statement can be a lengthy one. When you compile, this action is also done, but once only. For example, if you have a statement like the BASIC:

```
FOR N%=1 TO 50:PRINT J:NEXT
```

then the machine code for the PRINT action will be looked up, checked and then used 50 times rather than being looked up once, stored and then used 50 times as it would be in a compiled program. The ease of use of the interpreter is paid for by slow running, and the high speed of the compiled program is at the expense of easy checking and changing.

If you want to write programs for sale to others, however, the use of a compiler is a considerable advantage. For one thing, it's usually easier to protect machine code programs against copying (if you must – personally I never buy a program that I can't back up). More important is that you retain some control. Since you keep the 'source-code', the text that was compiled to make the machine code, you can alter the text and issue new versions. Others, lacking this source code, can't quite so easily modify your program. The main advantage, though, is that a compiler allows you to write programs in a high level language, but operate with fast-running machine code. For learning C, an interpreter is very useful, but for writing serious programs in C, a good compiler is essential.

C features

Unless you know more than one programming language, it's always very difficult to explain what there is about a new language that makes it useful. It's rather like explaining colours to someone who is completely colour-blind. Unless you have some notion of what *can* be possible, it's difficult to imagine how one language differs from another.

This is particularly true if you have only ever programmed in BASIC, because BASIC was devised as a beginners' language, modelled on one of the oldest languages, FORTRAN. If you have used a very advanced form of BASIC (meaning one that isn't really very much like BASIC!), such as BBC BASIC, then the task is easier. If you have used a 'structured' language like PASCAL or COMAL, then it's easier still. Even if your only other language is assembler, it still becomes easier to understand some of the ideas of C. The advantages of C can be summed up as structure, compactness and portability.

Structure is the idea that is most difficult to get over to many BASIC language programmers. This is mainly because of the way that BASIC is learned, often more by accident than design. A structured program is one that has been rationally designed using 'structures' like statement groups, IF..ELSE decisions and WHILE loops. Any program is a solution to a problem, whether it's to analyse the tax affairs of a business or to write a letter to your Uncle Toby.

In a structured program, the solution is easy to follow, because the problem has been split into manageable sections, and each section is dealt with by similar methods. An unstructured program, by contrast, is very difficult to follow because you can't see the flow of the argument, the process of solution.

Most high-level computing languages are almost unuseable unless you have some structure about your programs, and it's BASIC that's the big exception here. You can write a program in BASIC which starts with printing the answer to a problem, then inputting the quantities that are being processed, and then by processing bits of the solution in any order. How do you make it all work? Easy, just stick in GOTOs until it does. The result looks a mess, is often full of quite unexpected bugs, and is almost impossible to extend, improve or adjust. BASIC programming doesn't have to be like that, and even in a version of BASIC that is fairly primitive, it's possible to write programs that have some structure, making a lot of use of subroutines.

The trouble is that BASIC was not designed as a structured language, and most versions lack the statements that could make it a really structured language. In any case, if it had all these things, it wouldn't really be BASIC, it would be something like COMAL! Unlike BASIC, C is structured, but not to the point where you yearn for a couple of quick GOTOs just to relieve the monotony.

Along with structure goes modularity. A modular language is one in which you can break down any program into small units, or modules, and work on these *independently*. The main feature of a modular language is that these modules can be independent of each other and of the main program, which implies local variables.

To show what I mean, imagine a BASIC subroutine which is called up by the statement GOSUB 1000, and which uses the variable A%. If A% is used in the main program, then the subroutine can also use this variable name, and can alter the value that is assigned to it. This altered value can then be used in the main program. When you design a BASIC program then, you must organise the design of the subroutines very carefully, specifying what variable names must be used.

By contrast, in C, all the variables in the equivalent of the subroutine (called a function) are local. If you have a variable called A in a function, it exists only in that function. It does not have the same value as a variable called A which exists in the main program, nor can it affect that value.

It's only by very deliberate programming steps that you can pass a value into a function or pass a value back to the main program. This makes it possible to design each module separately, without having to worry about what variable names will be used in the main program, what values they have, and what will happen if these values are changed. Each module is a little island unto itself, and you have complete control over immigration and emigration. That's modularity!

The third feature of C is portability, which means that programs written with one machine can be used on another. No-one is likely to claim that BASIC is a portable language, because a program written for the BASIC of one computer is most unlikely to run on another machine without a lot of modification. I don't mean simply that a disc made by one machine will not run on another, because unless you use 8" disks that is a normal situation. The problem of portability in BASIC is that even listings for one machine cannot be used on another.

It *is* possible to write programs in a subset of BASIC, meaning a few commands that are common to all dialects of BASIC, but programming in this way is like trying to write in English using only the most common 50 words. By contrast, if you have a listing of a C program, then it should run, given a few conditions, in almost any machine that can make use of a suitable C compiler. That's portability!

The conditions are fairly obvious ones. The compiler must allow all the normal statements of C. Some compilers for small computers do not allow for 'floating-point' numbers, that is , numbers that include decimal fractions. If a program is intended to use graphics, then it's likely to run only on machines which support the same graphics structure. In the main, though, it's the compatability of compilers rather then the compatability of machines that matters. What you can be sure of is that your version of C will be portable for writing systems programs. This is the job that C was originally intended for and it's what C does best. In addition, if you happen to be using a machine that runs the UNIX operating system, then your C programs match your operating system like bacon matches eggs.

If you have only ever used microcomputers, then the main operating system you will probably know about is CP/M. The operating system for a computer is the part of the computer's essential ROM or RAM which does all the donkey work. The operating system must offer you an editor which allows you to type and edit your programs. It should supply input and output routines so that you can make use of disk drives, printers, and VDUs. It should have some kind of monitor so that you can find what is happening at machine code level, and it should provide a set of useful tools, utilities that can make programming easier. CP/M is one operating system that provides for these items, UNIX is another.

The distinguishing feature of UNIX is that it was originally designed for very large computers, and so it needs a lot of memory. The editor part of UNIX, for example, needs more code than the whole operating system of most small computers.

UNIX, like so many operating systems, evolved (in the mid-seventies) rather than being designed from scratch. Because existing languages were not really ideal for designing editors and other programming utilities, one man, Dennis Ritchie, devised a more suitable language - C. This doesn't mean that C was designed from scratch, as we've seen. Dennis Ritchie aimed at making *his* language a more general one than was strictly needed for extending UNIX, yet one which was still compact, reasonably easy to learn, and very satisfying to use.

If you had to make a quick summary of what C had to offer, you could make it in this phrase- *C is a language which combines the power of machine code with the structure of a high level language.* In longer terms, C allows you the control over what a computer does that you normally associate with machine code, and will generate compiled code that is almost as compact as machine code from an assembler. At the same time, though, C provides all the structures of a good high level language, like the facilities for creating loops, many different variable types, structured variables like arrays and all the rest. If you have only worked with BASIC, a lot of these advantages will be almost unimaginable. You can, however, fairly easily grasp another C advantage, the compacting of code.

It's possible to write C in perfectly correct form - and then to make the program considerably shorter by making some adjustments. Most of the statements of C can be written in a way that is not just shorter to read but which actually compiles faster and gives faster-running code! In BASIC, you probably know dodges like this, such as using integer variables in FOR...NEXT loops, using short variable names, omitting words like LET if they are not needed, and so on. These are just minor dodges compared to the ways in which a C program can be trimmed and turbocharged.

Another major difference is in the number of reserved words of the language. The tendency in recent years has been for bigger BASIC interpreters, allowing for 130 or more reserved words to be used. When you use C, you'll be struck by how few reserved words exist. The difference is that C is a small language with few reserved words. Instead of making the language contain a huge number of words for actions, C contains only the minimum necessary.

The other actions that you might need are supplied in the form of subroutines, or 'functions' as they are known in C. You are not stuck with the library of functions that you get with your C compiler, either, because you can buy additional libraries, and also add functions for yourself. It's because the core part of C is small that the language is so portable, and the library of functions is what makes C so useful. It's rather like having a BASIC of 20 reserved words, but several hundred subroutines on a disk that you could merge into your programs. To use C to advantage, then, you have to know what is in your own library, and how you can add to it.

Finally, not all versions of C are full versions. Several C compilers are subsets and the usual omission is floating-point numbers. If you are using C only for writing a word-processor, or for many types of system utilities, you can work in integers, and the lack of floating-point numbers is no loss. Several of these subsets, however, restrict integers to the familiar range of -32768 to +32767. This is what is called a 'short integer' in C, and it's much more useful if the compiler also allows 'long integers', range -2147483648 to +2147483647. This will look familiar if you have used the BBC micro, because it's the type of four-byte integer that the BBC machine uses.

If your requirements, however, are writing spreadsheets, or writing programs for scientific or engineering work, then nothing short of full floating-point arithmetic is suitable. Be careful, then, in your choice of compiler. If your compiler is one that supports integer only, it might be possible to have it upgraded later, but you can't be sure when. It will be useful for learning the language but will have to be discarded later. If you intend to earn a living with C, then go for a full compiler right away. It may be a lot more expensive, but if programming earns you money, you'll soon recoup the extra expense.

Chapter 2
Principles

Every programming language is based on a few important principles, and C is no exception. In BASIC, you are used to the idea that your program uses line numbers, and that statements are executed in order of ascending line number unless a GOTO or other redirection decides otherwise. You can forget most of the principles of BASIC when you start work with C, because the language is quite different. For one thing, you don't use line numbers. The main part of the program is written with everything in correct order, so that line numbers are not needed. Instead of calling a subroutine with a line number (like GOSUB 1000), you call a function with a name (like **sortit**). At first sight, this makes a program in C look more difficult to follow, like a country road with all the signs removed. We'll look in more detail at all this later on in this in the following Chapters, but for the moment there are two other important points about C.

One is that it can use compound statements, meaning a collection of statements that behave like one statement. In BASIC the nearest you get to this is a collection of related instructions gathered into one line with colons separating the parts, something like:

```
100 INPUT A:PRINT A:PRINT#5,A
```

in which all the actions concern the same variable. The other point is that C contains a lot more methods of controlling the flow of a program. There are extended IF tests, for example, using ELSE.

There are several different types of loops, allowing you to test for the ending condition of a loop at any point in the loop, to jump cleanly out of a loop, or even to ignore one pass through a loop and go on to the next. These are the 'structures' that make C a structured language, and these are the items that make C so unlike most versions of BASIC. Once you know what to look for you'll find that a program written in C is, in fact, easier to follow than one in BASIC. The most important difference, though, is in structure.

Program structure

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

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

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

This could **never** be mistaken for a program line written in C. For one thing, C lines aren't numbered, although some C compiler editors use line numbering for your convenience. If you write the lines in the correct order, the order that the compiler will deal with them, there's no need for line numbers.

The second point is that the subroutine starts somewhere later in the program, at line 5000. We can't have such a thing in C, because the compiler can't use line numbers. Instead of using a subroutine which is called by its line number, C uses a **function** which is called by using its **name**. Users of the BBC Micro and the QL are familiar with this idea, and the principle is developed much further in C . Even the instruction word PRINT is not used in C, and the semicolon does not mean 'don't take a new line'. Is it really true that knowing one computer language can prepare you for another?

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

```
10 DEF FNVat(S)=S*.15
20 INPUT "BASE PRICE";X
30 PRINT"VAT IS ";FNVat(X)
```

because the BASIC interpreter can't look ahead for FNVat. The similarity here is that a defined function in BASIC is called into action by using its **name**, FNVat in this example, rather than by using a line number. In a C program all functions must be defined, but this can be done following the main program or ahead of it, provided we follow the rules of C. Like a well-structured BASIC program then, which might consist of a core program of perhaps ten to a hundred lines of main program followed by subroutines, a C program is written in the order of **main**, then functions. This means that details, such as declaring variable types, all come at the start of the program, followed by the main program, and the functions often come last of all. This is a very logical arrangment as far as the programmer is concerned and it makes the writing of programs much simpler than is the case in other languages.

Any modern version of C can be expected to possess a good editing system, so it's easy to add statements at the start or at the end of a program if you have left something out.

The structure of a program in 'C' follows the principle of 'top-down' programming, often used as another name for structured programming. The principle is to break a problem down into outline and detail. The outline shows the main steps that are required to solve the problem, and the order in which these must be carried out. The details are then worked out for each main step, and this may in turn lead to another set of details. The 'top' of the problem is the outline, the bottom is the finest detail. The way that 'C' is designed encourages you, almost forces you, to construct your programs in this form. As you progress through this book you will see how this idea operates, and this, along with some experience of using 'C', will give you a much better idea of what is involved than any amount of reading about it.

Another part of 'C' that is tied up with structure is the use of the library, something else that we'll look at in detail later. The 'C' language is a small one, with very few reserved words as compared to any modern BASIC. This is because structured programming hinges on the use of functions to carry out the details, and these functions can be recorded on a disc or set of discs. These discs are the 'C' library, with each function acting as an extension of the language. Just as you can put together a BASIC program by using its keywords, you can put together a 'C' program by making use of library functions. In addition, though, you can extend the library for yourself with your own functions, so extending the language. This library system is possible only because 'C' is modular, and because functions can be written with purely local variables.

The order of things

C is not just a structured, modular, portable compiled (or interpreted) language, it is a language in which programs can be written which will compile to unusually compact and fast machine code. Once again, this is possible only if you write your program in the correct order. The most important idea to get used to, if your programming experience is in BASIC, is that types of variables and their names have to be **declared** before they can be used. In BASIC, you can write a line like:

```
100 A%=5
```

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

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

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

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

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

```
NAME="SINCLAIR"
```

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

To a BASIC programmer, it seems odd that you might find a variable called NAME and not have any mark (like $ or %) to tell you what type of variable it is. Once you get used to C, though, you will find that it's more natural to start off a program with a list of the variable names that will be used. This is another aspect of structure –you need to have planned these names in advance rather than just put them in at a moment's notice. It's only too easy in a long and poorly planned BASIC program to use a variable name twice without realising it, and so causing a very obscure bug. You can't do this so easily in C.

First steps

All C programs can be constructed in very much the same way, and though you can leave out some steps in certain versions, it's advisable to keep to the rules of standard C . If you do, it's much easier to write C programs for practically any machine. In addition, it helps a lot if you write programs the way that you ought to design them, outline first and details later. Remember also that you need to keep to the rules for the machine you are using as well. Any features of a particular machine or compiler that are peculiar should be listed prominently in the manual that comes with the compiler, and it's likely that you will know the peculiarities of the machine you are using in any case.

One simple example is the HiSoft compiler for the Amstrad machines, which allows the use of line numbers. If you type a line number and then a statement, with no space, this will be rejected by the compiler, because the Amstrad machines all require a space between a line number and a statement.

Getting back to program construction, though, the layout order of a program in C is one of definitions, main function , then functions that are called by the main function. The main function is marked by the use of the word main(), and the extent of this main function is shown by the use of curly brackets, one to open the main function, and the other to close it. The outline of a C program is therefore:

Various definitions

main()

{

Declarations for **main**

The main function.

}

The functions that are called by the main program or by other functions.

and in this brief outline of a program there is quite a lot to assimilate. To start with, there can be quite a lot of material preceding the main function. You can, if you like, put all the other functions before main(), and there will be some functions that **must** be put before **main**, unless some alterations are made to the way in which these functions are called. In addition, definitions are put in here.

A definition in this sense refers to a word that will have a special meaning in the program. You might, for example, be reading files in your program, and so you will need to define what is meant by EOF, the end-of-file marker. On most machines, this is -1 and so you would define EOF as meaning -1, and in your program use EOF wherever it was needed rather than -1. This has two advantages. One is that if you have to adapt your C program to a machine that uses a different value for EOF, then the change is done by editing one line, the definition line. The other advantage is that it's a lot easier to see what a program is up to if it uses abbreviations like EOF rather than numbers like -1.

The next point is that the word **main** needs to be followed immediately (no spaces) by a pair of round brackets. These brackets are essential and cannot be omitted. The reason is that each function in a C program uses brackets to contain the variable names of values that will be passed to the function. You quite often don't pass any values to main, but main is just another function and is constructed in exactly the same way as any other function. There may be times when you do need to put something between the brackets of main. This will arise when you call a program and at the same time pass some values to it. Not all C compilers or operating systems provide for this action, but because main is a function like any other, the brackets are part of the **main** title.

The next feature is the set of declarations that apply to **main**. These declarations will include each variable name that will be used in main, and possibly some of the variables that will be used in the other functions. The difference is one that we'll look at later, between global and local variables. As well as declaring the names that will be used, the type of variable must be declared. We'll look at variable types shortly, but the principle is that you have to state a variable type, and then list what names are of this type. If you are using integers, for example, you declare the type integer, and then list all the names that you will use for integer variables. Some of the variables that you declare in this section may be far from simple, like arrays and structures, and we'll look at these items later.

The main function then starts with a curly bracket (or *brace*), and will end with the opposite facing curly bracket. Between these two brackets you may have various statements, and these may use further curly brackets, so that the brackets will be nested. As in any nesting, you have to ensure that the nesting order is correct, and that there are as many closing curly brackets as there are opening curly brackets. This should not be difficult to ensure if your programs are correctly written. In a correctly written C program, the actions of the program should be split into small groups, each of which can be carried out by a function. The main function therefore should be short, and since it's short, it's not difficult to check the balance of any nested brackets.

In general, if any piece of a program takes more than a page to list, you haven't really planned it too well. The whole point of having a modular language is to work in modules that are of convenient size, and a page is convenient in the sense that you can take in a page at a time. You should never, using C, get the feeling of helplessness that you get when you are holding 3 yards of listing in BASIC and you find that the first line is:

```
    10 GOTO 10515.
```

Now, having looked at the outline of the construction of **main**, what does each other function look like? The answer is 'almost identical', and only the name is changed, because no other function can use the name **main**. If your function name is **getinput** for example, then the function will have this name, with its round brackets following. Within these brackets might be the names of any variables that will be used inside the function, so that a function called **getinput** might appear as

```
getinput(name1,name2)
```

where namel and name2 are variables that will be used inside the function. Following the title, these variable names must be declared in the usual way, then an opening curly bracket announces the start of the function. More variables might be declared at this stage, and these would be variables that were not being used to pass values to the function. The actions of the function follow, and if any variable value has to be returned, then this is done using a **return** statement. This is **not** like the RETURN of a BASIC subroutine, though it has the same effect of returning control to wherever the function was called from. For one thing, the return of C specifies a variable to be returned; for another, the function will return on its own even if there is no return statement. The point at which a function returns to **main**, or to whatever function called it, is then marked by the closing curly bracket.

As you can see, the construction is almost identical to that of **main**, but in the case of another function there will nearly always be some variable names between the round brackets. These names represent values that the function will work on. If you have only used subroutines in BASIC, this is something that will be very new to you. Suppose, to take an example, that you had a program which dealt with words, and at some point in the program you wanted to call up a routine to put a word into capital letters. In Microsoft BASIC, you might represent the word by the variable name WD$, and you would use a subroutine which changed the contents of WD$. This new value of WD$ would then be available to be used in the rest of the program. In this example, WD$ is a variable that is used in the main program and in the subroutine, its value is available to both, and it can be changed in the subroutine. It is, in short, a *global* variable.

Things could hardly be more different in C . We can create global variables, and an illustration will be dealt with later, but we seldom choose to use them. Unless we choose to make use of global variables whose value is maintained in all sections of the program, no quantity that exists in the form of a variable in the main function can be used in a function unless the value is passed over to the function. Being passed over means that the value has to be held in a variable name which is put between the round brackets in the function name. For example, if we used the variable name of **word** to hold a string of letters, then this could be used in a function **makecap** only by calling the function in the form:

```
    makecap(word)
```

if **word** is omitted, then the function has no argument, nothing to work on. There is another difference. If the function makes some changes to the value of **word**, this will not necessarily cause any change to the value that is held in the main program. For example, if **word** represents "Figment", and this is changed in the function to "FIGMENT", then after the function has run, the variable **word** is still likely to contain "Figment". If the value of a variable has to be changed by a function, then rather more specific methods have to be used.

As far as each function is concerned, it operates in a little world of its own. Whatever quantities are passed to it, it can use, modify, print and perhaps pass to other functions, but this has nothing to do with any variable of the same name in the main program. A variable like this is called 'local', and if you have programmed the BBC Micro you will already be familiar with the idea. C takes the idea of local variables much further, though, because in C all variables are normally local- we try to avoid global variables as far as possible. The purpose is a very good one - it allows you to design a function quite independently of the main program. Each function will be designed with its name, the name of variables it will use (not necessarily the names that are used when the function is called by the main program) and its own set of actions. One person can design a main program, a different person can design each function, and yet the whole program can easily be put together with the minimum of fuss. Try that with BASIC!

This is what makes it possible to have a function library with a C compiler, or on a separate disk. Each function is independent. If you have a function which is called **toascii(c)**, and the c within the brackets means a single character code, then you can call this function from your main program with the words **toascii(first)**, providing that you have declared and used the variable **first** to mean a character. It's not the name that counts, it's the position in the brackets, so that **first** in the calling statement gets treated as c in the function definition. If the function uses several variables, then the order must be the same when you call the function. For example, if you have a function **convert(a,b,c)**, and you want to use this on variables **J1, POINT2** and **RANGE**, then you have to call with:

```
    convert(J1,POINT2,RANGE)
```

which means that in the function, **a** will be used to carry the value in **J1, b** to carry the value in **POINT2,** and **c** to carry the value in **RANGE**. The order has to be maintained, and the variables have to have been declared as the correct types. This type of action is called 'passing by value'.

Data types

In BASIC, you are accustomed to three primary data types (types of variables) and usually one structured data type. The simple data types are integer number, floating point number and string. In BASIC, these three different data types are represented by names, with distinguishing marks used for integer and string types. For example, you could use the name AB for a floating-point number, AB% for an integer, and AB$ for a string. These are called the primary data types because all the forms of data that you use must be expressed by one or other of these variable types. The only structured data type in most varieties of BASIC is the array. You can, for example, use the names AB(0) to AB(10) to represent floating-point numbers in an number array. These numbers will be stored consecutively in the memory, which makes it easy for the computer to find any one member of the array provided that the address of the first member AB(0) is known.

Most varieties of BASIC allow string arrays as well, so that you can have arrays like WB$(0) to WB$(50) used in your programs. The reason for using the name 'structured' of an array is that there is a structure of values in the memory of the computer arranged so that the machine can get hold of any one value easily.

In C, there is a much wider range of both primary and structured data types. Like BASIC, we can have floating point numbers and integers, with the difference that the integers can be short integers, stored in two bytes, or long integers stored in four bytes. There's also a long version of the floating-point number, called the double precision number. As the name suggests, a double precision number is stored using twice as many bytes as the ordinary floating point number. This makes arithmetic much more precise, but it slows program action down considerably. You would normally use double precision numbers only for some accounts work, and (mainly) for scientific or engineering programming, but in fact most compilers will follow the C rules and use double precision for any arithmetic using floating point numbers. That accounts for four primary data types for numbers alone.

After that, things start to be very different. There is nothing corresponding directly to a string variable in C. The data type that we use when we are working with words is the character, which corresponds to a single ASCII code in BASIC. Since a character is an ASCII code, it can be stored in one byte. When we need to store a word or several words, we use an array of characters. This kind of thing has also been used in some versions of BASIC, such as in the older Atari machines.

There is a type of variable, the pointer, which performs much more in C than in BASIC. A pointer is an address for a variable rather than a data-type, so it's a two-byte integer for machines that use 8-bit microprocessors. The value of a pointer is that it needs only two bytes to refer to any other kind of variable, character or number, integer or float, single or double. Even more usefully, a pointer can refer to an array by holding the address of the first member of the array. The pointer is the means by which we can make a function change a quantity in the main program, and it's also the way in which we can tie ourselves into contorted knots if we're careless. We'll leave details of pointers until later (Chapter 7), pausing only to note that Microsoft BASIC uses pointers in the VARPTR form and Amstrad Locomotive BASIC uses them in the form of the @ prefix with a variable name.

The structured data types of C are the array, the structure and the union. The array is arranged very much like the array in BASIC, and you can have an array of any primary data type. The character array is rather special, because it's the equivalent of the BASIC string variable. For that reason, character arrays are used to a considerable extent. The other types, the structure and the union have nothing remotely corresponding in BASIC, and once again, we'll leave them until later, Chapter 10. The most common and useful data types are the character, the short integer, and the character array, and 95% or more of your programming in C is likely to use these types. All C compilers will be able to cope with these data types; even the subset compilers which are cut-down in the sense that they do not permit floating point numbers.

Getting started

To get started with any variety of C , you will need a compiler program, of which you have a large choice for most micros, with the exception of the BBC machine prior to the Master-128. For machines that run an operating system like CP/M or MS-DOS, the choice of C compilers (and interpreters) will be almost embarrassingly large. If the operating system of the machine is a manufacturer's special, you may find that there is no C compiler available, in which case hard luck! Nowadays, however, with more attention being paid to serious programming on small machines, there aren't many machines which have the memory to cope with C , a useful disc system, but no compiler available. When you load the compiler, there will be a small menu of choices available to you, of which you will take the edit mode when you start to write programs.

In general, C compilers allow you to create or edit text, compile it into code, and run the code. This doesn't mean that you can do all of these actions while you are running the compiler. For most compilers, running code is a separate action, and you must store the code on a disk, then run it separately. This can make developing a program very slow work unless you design your program properly in the first place. If you have been used to the try-it-and-see style of programming in BASIC, then you will have a number of bad habits to eradicate. You can't just stop a C program at an error, make your corrections and then carry on. You can't even find out easily what values your variables had at the time when the program crashed.

If you need to change the program, you have to load the compiler in again, load in the program text, change it, recompile and then run again. There is a strong incentive here to get things right first time, and the principle of programming in sections really starts to make sense, because you can work with sections that have been tested and proven. A few compilers allow you to keep the text and the machine code in memory at the same time, so that checking and correcting can be done more quickly. Obviously, this applies only to short programs, because the compiler, the text and the machine coded program must all sit in the memory at the same time. The facility is very useful, though, because C programs are normally written in short chunks, and to be able to write, test and debug all in one loading is very handy.

For longer work, such compilers normally allow the compiler to take text from a disk and put the resulting machine code back to the disk. Another possibility is the provision of a linker that will merge a number of machine code sections into one program. If you have a choice of compilers, one that can optionally work with compiler, text, and machine code all at one time has a lot to commend it.

Suppose then that you have a compiler loaded, and in edit mode. For some C compilers, the editor might be a separate unit, and if you were using UNIX, there would be an editor as part of the operating system in any case. Whatever you have, you type in your C listing to create a text file. A text file is exactly what it is, a set of ASCII codes that can be recorded on a disk. Unlike the BASIC interpreter, which converts keywords into special one-byte codes, the text file is just the form of file that you would get from a word processor.

A few C compilers, in fact, are sold with no editor, assuming that you'll have a word processor that can be used for creating this text. With the text recorded on disk, the compiler can then be invoked. Once again, in some varieties of compiler this will involve replacing the editor with a compiler.

During compiling, the compiler may need to read functions from a function library disk. This makes it handy to have a twin-disk system, or to have the library on the same disk as the compiler. The need for the library is brought about because even quite straightforward actions like comparing two strings, or finding the length of a string are carried out by library functions, and the code which carries them out must be read from the library disk as compiling proceeds. Some compilers force you to read in all of the library functions into memory, others allow them to be picked off the disc as needed. Whichever method is used, the incorporation of library functions can make compiling slow in some cases, and you will need to find from your compiler manual if there is any way of speeding this up. Very often it is possible to copy library routines on to another disk, so that the few routines that you need to use many times over can be kept separately rather than in company of a large number you hardly ever use at all.

At the end of compiling, unless errors have been found in your text file, you will have a machine code program on a disk. For small machines with a single disk drive, the compiler will have arranged for you to change disks when the code is ready, because its unlikely that you will want it placed on your compiler disk. For twin-disk machines, you will have to place the compiler/library disk or disks in one drive, and the output disk in the other.

A few small compilers may keep the machine code in memory, and will not record it unless you follow a routine that is laid out in the compiler manual. The compiler will have created not only the machine code but a 'run-time' system. In other words, you will have code plus the means of running it in the machine when the C compiler is not in the memory. This may involve using an instruction like CALL 2026 or SYS49110 when the machine code has been loaded into the computer. This machine code program can be loaded and saved like any other machine code for that machine. If the C compiler created code for CP/M, then the machine code that you produce will be able to be adapted for other machines that run CP/M as well, considerably broadening the base of your programming.

The same sort of thing applies to a C compiler which produces code for use with MS-DOS, because a huge range of machines will then be able to read the code, and most will be able to use the same disks thanks to the dominance of the IBM PC standards.

Chapter 3
The C Program

Starting simply

So far, you may have concluded that you now know a lot about the principles, but precious little about how you actually get your teeth into the C language. That's inevitable when you start to learn any structured language, because until you know the reason why things are done as they are, learning the language seems like a set of pointless rules, and when you try to write programs for yourself, you keep getting error messages whose purpose baffles you.

Error messages in most C compilers are not exactly what you are accustomed to in BASIC. Very often, a compiler does not pick up an error at the point where the error exists, so when your compiling stops with an error message, the error may not be in the line that is displayed on the screen. Worse still, the error message may not be particularly helpful in finding the error. Some compilers are distinctly more helpful in this respect than others, but in general if you try to learn C in the way that you probably learned BASIC (try it and see what happens), then you are in for a lot of frustration. Bear with me, then, while we get into the subject gradually.

From what we have already looked at, then, the simplest possible outline of a C program then, is:

```
definitions
main(){
declarations
statements
}
```

and we now have to take a look at this in rather more detail. One point of detail that we need to look at now is the C statement. In BASIC, a statement is an instruction like PRINT A which might be put into a line, or made part of a multistatement line. The end of a BASIC statement therefore is marked either by the use of a new line or a colon following the statement. In C , the end of a statement is marked by a semicolon, and we have to be careful to ensure that the semicolons go in the correct places. In BASIC, the semicolon is used to ensure that printing is to be kept on the same line, and the use of the semicolon in C takes a lot of getting used to. In C you can, for example, use the semicolon to separate statements in the same line, as you use the colon in BASIC. Omitting the semicolon is a way of instructing the compiler that there is more of a statement to come, the kind of thing that you might find in a test, or a loop, for example.

Another point is that you can construct compound statements. These consist of a set of statements that start with an opening curly bracket and end with a closing curly bracket. A compound statement like this is treated as one single statement by the compiler, just as a multistatement line using colons is treated as a single line by a BASIC interpreter. At this stage, it's a bit pointless to describe the rules about semicolons, because until you have had some experience in writing programs, you won't really see why semicolons are used in some places and not in others. For that reason, I'll point out in each of the early examples the few instances in which a semicolon has not been used where you might expect it to appear.

In the example of program outline, the first line consists of the special name **main()**. A C program consists of a set of named **functions**, using whatever names you like to give them, but there must always be one that is called **main**. The brackets are an essential part of this, and it's not very often that you need to put anything between these brackets. For other names of functions, though, you will usually need to place variable names between the brackets. In any case, you can't omit them. The **main** program is the one that calls up all of the other functions, just as a BASIC core program can call up various subroutines. The curly brackets then show the start and the end of the **main** program, with the { indicating the start, and the } the end. You don't need any other way of marking the end of the program, and the word END is not a reserved word of C and should not be used. When you want to read a C program, then, you look for **main** and then read what is enclosed between the curly brackets. If the program is well laid out, taking one line for each statement, and indenting lines that form part of a loop, the program should be very much clearer and easier to understand than one in BASIC.

The way that a program is typed is of much more importance in C than it is in BASIC. One important point is 'white-space' meaning spaces and also the TAB character and the newline, all of which cause a white space to appear. Whitespace characters are used in 'C' as separators, and you cannot run words together in the casual way that so many varieties of BASIC allow. As you'll see, some 'C' functions are very fussy about whitespace characters and this can be a source of problems unless you realise that, for example, one function might be arranged to skip over whitespace characters, and another to look specifically for them. When you use an INPUT in BASIC, you never think that pressing the RETURN key adds another character to whatever you have typed, because BASIC isn't organised that way. In 'C', though, this is important - the RETURN key gives a whitespace which is at the end of whatever else you typed.

Indentation is also important, and it should conform to standards. The general rule is that groups of lines that belong together and form a subsection of a function should be indented. For example, all the lines that constitute a loop should be indented by one space from the lines of the program that contains them, and if loops are nested, then the indenting should be nested also. Indenting should make it easier to pick your way through a program listing, and it usually does, though if there are a lot of nested levels, the appearance can be unsatisfactory when lines have to be indented so far that the ends spill on to the following line. In general, though, if you need that level of nesting and hence indenting, your program is probably not well constructed, and will certainly run very slowly.

As in BASIC, you can sprinkle reminder lines about your program. This is very often omitted in a BASIC program because each REM in a BASIC program has to be read by the interpreter even if it is not going to be acted upon. For a compiled C program, however, the reminder lines are only part of the source-code, the ASCII text, and they don't exist in the machine code (object code) which is the bit that actually runs. You can therefore afford to be quite generous with your reminders, subject to the amount of memory that will be needed for the text.

The reminder in C is marked with the combination of a forward slash and an asterisk with no space between them. Unlike the REM of BASIC, you have to mark both the beginning *and* the end of the remark. At the start of the remark, you use /* and at the end you use */, and you have to be careful to get the order correct. You also have to distinguish the forward slash (/) from the backslash (\), because the backslash is used for very different purposes. A reminder, then is written in the form:

```
    /*this is a reminder*/
```

and you would normally put it in a line by itself, or following the end of a statement.

The simplest possible program is then written using keywords. A keyword in C is rather like a keyword in BASIC, it is reserved for a special purpose and you can't use it for anything else. Keywords are generally typed in lower-case and must be correctly spelled, otherwise an 'Undefined symbol' error will be announced after you have compiled and when you try to run the program. This means that any of the errors which you would think of as 'syntax errors' in BASIC are very often not discovered in a C program until after compiling. This wastes a lot of time, so you need to be rather careful about checking what you type. Figure 3.1 shows the keywords that are used in most varieties of C compilers .

auto	specifies type of variable or function.
break	break out of loop.
case	marks a choice made by using switch.
char	character variable type.
continue	go to start of loop.
default	select default option in switch.
do	start of do..while loop.
double	double precision variable.
else	alternative in if statement.
entry	not implemented, reserved for future use.
extern	declaration for global variable, particularly for use
with	linked sections
float	floating-point variable.
for	start of counter controlled loop
goto	jump to position of label name.
if test	word, used along with a test.
int	integer variable type, range -32768 to +32767.
long	double size variable type.
register	variable type, not implemented in microcomputers.
return	pass back value of parameter from function
short	normal variable type.
sizeof	measuring number of bytes in variable.
static	type of variable using fixed memory.
struct	compound variable consisting of several fields.
switch	passes control to one of a number of statements.
typedef	defines a name as meaning a variable type.
union	variable type which can be one of a defined group.
unsigned	number in range 0 to 65536 (short).
while	marks start of while loop, or end of do loop.

3.1 The standard set of keywords for a C compiler, with brief reminders of their action. This list looks very brief as compared to a list of BASIC reserved words.

Note that versions of C for specific machines will have a few extra keywords, like **inline** and **cast** in HiSoft C for the Amstrad machines, and others, notably the C for the IBM PC, use more keywords for random access disk filing. A few compilers will omit some of these keywords, but in general, you will find all of them in every C compiler.

The word **main**, note, is *not* one of the keywords. This doesn't mean that you can use it for anything you like, but it is a title for the main program, not a word which describes an action. This is an example of a word which is an **identifier**. In BASIC, the only identifiers that you use are filenames and names for variables. C uses a lot more types of identifiers, and they are used in much more interesting ways. In this case, the word 'main' identifies the main program, and you can use other words to identify the functions (the C replacement for subroutines) which are called up by the main program. You also use identifiers for other things, like variable names, subject to a few rules.

The rules are that an identifier must start with a letter, upper-case or lower-case. Many compilers treat upper-case as being different from lower-case; some provide for automatic case-conversion (particularly compilers that compile to code that runs under CP/M), and a few, mine included, treat upper-case and lower-case as being identical. Since you must use lower-case for keywords, it makes sense to stick with lower-case for identifiers too. Professional programmers use upper-case letters in identifiers which are present for special purposes. You can then follow this first letter with other letters or with digits, but no blanks or punctuation marks.

The only character that is allowed, apart from letters and digits, is the underscore (_), assuming that there is one on your keyboard. If your machine has no underscore character, a compiler for your machine may permit the use of another character in place of the underscore. The underscore is useful as a way of making long names more readable (like name_of_item). You could, if you wanted to, start a word with an underscore, but here again, it's better not to, because this could lead to trouble later on in your C programming career. The reason is that words that begin with an underscore are often used for special purposes in C functions, and you may cause conflicts if you start to use them. You **can** use names of more than eight characters, but **only the first eight** characters will count in most compilers. This gives you rather less choice about things like variable names than you have in some varieties of BASIC, particularly BBC BASIC and the Locomotive or Mallard BASIC of the Amstrad machines.

Most C compilers will contain a few built-in functions with identifier names which are therefore already allocated. In addition, you will probably have a library of functions that you can call upon. A typical set, from the HiSoft compiler for Amstrad machines, is listed in Figure 3.2, and when you look at this list, you might think that this was another set of reserved words, as many of them would be in BASIC. The difference is important.

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

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

3.2 The built-in functions for Hisoft-C. This is supplemented by the much larger list of functions that can be read from the library.

All of these identifier names can be used by you for something else if that's what you want. If, for example, you want to call your program **gets** or **puts** then you can do so. You would be foolish to do this, because by changing the meaning of these names, you are losing the use of some action that you might need, but you will not cause any error message. The difference is important, because if you try to use a reserved word for anything else, the error will be signalled; if you use one of the 'predefined' identifier words, there's no error, and you won't be informed. You may wonder, however, why some action later turns out to be impossible!

Following the **main()** identifier , there will be a newline (obtained by pressing the RETURN or ENTER key) and the opening curly bracket which marks the beginning of any function. The next line will be the first statement of the program, and it will end with a semicolon. You will normally take a new line following each semicolon in order to make the listing look neater. This isn't enforced, and you can make the program one long chain of statements separated by semicolons if you want to, but if you can't read it afterwards then it's your own fault.

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

When we get to specific examples of programs, no matter how simple, it's important to try them out on your own compiler to get the feeling of the use of main() and the curly brackets. I know that the first few programs look ridiculously simple, but if you try some mistypings you will learn how your compiler deals with them. Find, for example, what type of error messages you get when you miss out either kind of brackets, or omit one of the terminators around a reminder line. Find also at what stage you get the error message. With luck, a lot of error messages will be picked up by the compiler, and only a few show at the run stage.

A few lucky people may have compilers which pick up the errors as you type them, and if you are using a C interpreter in place of a compiler, then you should have the same sort of error reporting as you are accustomed to in BASIC. By exploring the error messages now, you can save yourself a lot of trouble later, because if you come across such messages when you are working with a longer program, it's comforting to know why the messages are delivered. Without this experience, you tend to look in the wrong places and assume that what has gone wrong is very much more complicated than you think.

Storage classes

Once again, just as you are hoping to get your fingers on the keys, there's something else to learn about. Once again, though, it's something that would cause a lot of bafflement and dismay if you met it with no warning, because there's nothing like it in BASIC. The subject is storage classes, and it concerns the way that your computer stores its variables. I should warn you at this point that some of this topic betrays the fact that C was designed before microcomputers existed, so that some storage classes just don't exist on micros.

To start with, you don't normally have to think at all about how variables are stored in a BASIC program. When a BASIC program runs, its variables are stored in the memory which lies just beyond the program text area. As each new variable appears in the program, a new space is created, so that this storage is dynamic, changing all the time. This also explains why many BASIC interpreters cannot allow you to continue running a program after making changes. The changes you have made will have altered the length of the program listing, and altered items in the memory where the varables are stored. After editing, then, you have to run your BASIC program again from scratch to build up the variable values in the memory again.

C appears to allow rather more choice, by permitting various classes of variable storage. To some extent, this is an illusion, because the storage will normally be in the memory, and the main difference is how long it is held there. In a very complete version of C you might find classes labelled as **auto, static, extern** and **register**. On most microcomputers, the **register** class will be absent. As the name suggests, this allocates a register of the microprocessor for each variable, and none of the microprocessors used in small computers are ever going to have enough spare registers to allow this.

The register class was devised to allow for very fast-access to some variables in mainframe computers which used hundreds of registers in the CPU. Strike this one off your list, but check to find what your compiler will do if you forgetfully specify a variable as being of **register** class. If you're lucky, the compiler will allocate it to some other class and inform you. If you are very unlucky, it will cause a crash, prompting you to say unkind (but possibly true) things about the author of the compiler.

The most important storage class is auto and if you don't specify any storage class for a variable, this is what you'll get.

An automatic variable is one that is local inside the block where it is defined. By block, I mean any function of the program. If you define an automatic variable **filnam** in a function called **spotit**, then this variable exists only between the start and the end of function **spotit**. If **spotit** calls in turn another function **spillit**, then variable **filnam** can be passed to this function, and used in function **spotit** again after **spillit** has ended. When function **spotit** ends, however, variable **filnam** no longer exists and has no value. Since at least 95% of the variables that you will use need to be local like this, it makes sense to have this as the default. Using auto variables, which are stored on the stack, can be slow, however, and several compilers urge you to make more use of the third class, statics.

The static class of variable is an interesting compromise between a local and a global variable. Suppose you have a function that defines and uses the name **count**, and has specified that this variable is static. At the start of the function, **count** is initialised to 0, and somewhere else in the routine, **count** is incremented. When the function runs for the first time, then, count is assigned with the value 0, and then this is later incremented to 1. When the function returns to **main**, or to whatever function called it, there is no longer a variable called **count**. If this variable is static, however, the value it got first time round is stored. The next time the function is called, the initialising step will be ignored, and **count** is given the value of 1, ready to be incremented to 2. If the variable had been an auto type, then the action would simply have been to initialise to 0 and increment to 1 again.

Static variables retain their **values** after a function that contains them has ended, even though the *name* of the variable is deleted and can be used by a completely different variable in the main program. The use of static variables is handy if you want to count the number of times a function has been called, or keep track of which elements of an array are being used. In addition, some actions require the use of static variables.

If you want to declare and initialise an array variable in one step, for example, the array variable must be declared as a static one. Some compilers may insist that **any** variable that is to be initialised at the time when it is declared cannot be automatic. This takes some getting used to after BASIC, because in BASIC you normally declare and name in one operation, such as A$="NAME" or B=123.26. Once again, this is something that you have to sort out between yourself and your compiler. The best course usually is to use static variables only when you really need them, which is when you need to keep a value preserved between one call to a function and the next. If you follow this advice, you will seldom use statics, and that's a good course to follow unless your compiler manual urges otherwise.

The other class of storage is external, usually abbreviated to extern. This is the class of variable that you use (whether you realise it or not) in BASIC. An extern variable is a global variable, which hangs on to its identity and its value through any block in which it is defined. One way of defining an external variable is to define it and assign it before the start of a **main** program. This will ensure that the variable exists and retains its value in any part of the program **main** or any functions called within main. This looks attractive if you have been used to the way that variables behave in BASIC, but it's not really a good thing to have except for specialised purposes. For one thing, it destroys the principle that you can make your programs out of little sections that can be developed separately or called from the library. If you insist that each section shall work with an external variable called **univers** for example, you are making all your routines into specials, suited only for the program in which this variable is used.

The other deterrent is that when you are working with a long program, having a variable that appears all over the place is a nuisance. You have to be careful of changing the assigned value, for example, in case this should cause problems in another part of the program. Unless you need to link sections together, then, you should avoid external variables as you would a typhoid carrier. You will have to decide from the manual for your own compiler if there is any instance when the use of externals is essential.

The pre-processor

One feature of any C compiler is very much like a word processor action, and is certainly not like the action of anything in BASIC. This is, for historical reasons, called the pre-processor, because all the actions in this section have to be carried out before compiling can start. In this book I shall call it the definition section. A good reason for this latter name is that each line in this section begins with the word **#define**, and what follows it is a definition of something that will be used as a constant in the program. We have already met the idea that we might define the letters EOF to mean a number (usually -1) that is used to indicate the end of a file. This would be put in by the line:

```
    #define EOF -1
```

putting a gap between the three parts of the definition. Anything that might be used as a constant in a program should be defined before the start of **main** in this way, and common items for such definitions are CR for ASCII 13, TAB for ASCII 9, BS for ASCII 8 and so on. Even more useful is the fact that strings can be defined in this way, allowing you to produce messages in the program. For example, using:

```
    #define ERMES "Warning - error"
```

will allow you to produce the full error message by printing ERMES.

This use of #define is not the same as assigning a variable, because the item that is defined in this way is put in full form into the program listing in each place where it is used.

To make this clearer, suppose you had a string variable which was assigned as TXT. Now since this is a variable, it will be stored in some part of the memory, and when TXT is printed, the print routine obtains the memory address for the text, and prints it. If, however, you had used a define line such as:

```
    #define TXT "This is a message"
```

then wherever you used TXT in the program, the preprocessor action would put in the letters (in ASCII code form) of the message, before the compiling action started.

The difference between variables and these defined constants is rather like the difference between subroutines and macros, if you have met macros in assembly language. The #define lines are macros, pieces of text with abbreviations, and the action of the preprocessor is to work on the text so as to expand each abbreviation out into its full text. Your compiler may allow the #define lines to define actions as well as constants, so that you can have lines such as:

```
    #define SUMSQ(a,b) sqr((a*a)+(b*b))
```

which can be used in the program to get the square root of the sum of squares of two number variables. The simpler compilers may refuse to accept syntax of this type, and you will need to check carefully how your compiler deals with such lines. In most cases, it's very important not to have a space between the name of the defined word and the brackets that follow it, because a space is taken as the marker between the name and the item to be substituted. It's also important not to end such a line with a semicolon, because this would make the semicolon part of the substituted text.

The reason for the name 'preprocessor' is that in early compilers for mainfames there would be so many #define lines that a separate program was needed to process them before the compiler was used. This separate program was the pre-processor, the part that filled in the definitions, so saving time in compiling. Nowadays, most compilers include this action, and there is no separate preprocessor program, though there must be a routine that carries out the substitutions in the text before the compiler can get to work.

One of the most common preprocessor actions apart from #define is #include. This allows the contents of another file to be included in the compilation, and it is often found at the start of a program to allow a file of #define lines to be added. When your #define lines are used in every program you write, this is a very useful way of ensuring that they all get into place. Several compilers make use of #include as a way of gaining access to a standard library of routines. For example, it's very common to find that you have to use as your first line:

```
#include stdio.h
```

in order to include automatically the input/output routines in the library.

Other definition lines starting with the hash sign are more dependent on the compiler. For example, the HiSoft C compiler for Amstrad machines permits #error, #list, #direct (not on the CP/M version) and #translate, all of which have specialised uses. #error allows the full error messages to be removed from memory, releasing more memory for program files. #list can be used to enable or disable listing, and is switched by using the + or - sign. For example, if #list+ is used before a #include, and #list- following, the included file is not shown on the screen. The #direct command is not a normal type of preprocessor action: it allows a statement to be directly executed, and this is a special feature of HiSoft C. The #translate is used in the non-CP/M version of HiSoft Amstrad C to convert a program into stand-alone form, suitable for use in any machine with the Locomotive BASIC ROM. This implies that all the run-time routines are coded, something that is not normally carried out when the compiler is being used for small program segments.

Chapter 4 Fundamentals

It's time now to start some practical programming. If you feel that it isn't one moment too soon, then I sympathise, but you'll see when you get further into writing in C for yourself why I took so long to get to this point. Even now, we're not going to plunge into any very fancy programming. The main thing at this point is to establish how we carry out some fundamental actions in C , because unless you can write program listings for such elementary things as printing on screen and the input of values you are not likely to get very far with C . Some textbooks are notably bad in this respect - they plunge you right in, and expect you to cope. Softly, softly is the watchword here. It's fine to dive in if you are working in company, with an instructor to hand. If you're struggling with your own machine and relying on a book to learn from, then the slower the better. In any case, what follows is going to require you to know how to use your compiler, and you should have it loaded and ready.

Output to the screen

You have probably already noticed that C does not have a reserved word PRINT. There is, in fact, no reserved word for the action of putting something on the screen. This action is one that carries an identifier name for a library routine rather then being one of the reserved name actions. The identifier word that you need is **printf**. This word may be a library routine, or, as in HiSoft C , built-in as part of the compiler coding, so that the routine does not have to be read from the disk each time you compile it. You will need to find out at this point just what your own compiler will require. The use of **printf** is, however, quite different from the use of PRINT in BASIC. The name **printf** has to be followed in brackets with details of what has to be printed. This means not only what you want to print, but also *how* you want it printed, formatting as it's called.

The formatting commands and the items that you want to print are all included within brackets, with quotes around the formatting commands and any characters that are to be printed. What is 'written' on the screen in this way can be a number or it can be text. The simplest possible examples of text writing look sufficiently like BASIC to be easily understood when you are reading a C program. Figure 4.1 shows an example, which you can type, compile and run.

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

4.1 A simple listing showing how printf is used to place text or numbers on the screen.

In this example, the actions are of writing a word and performing a piece of arithmetic. The writing of a word is rather different to the PRINT "My message" that would be used in BASIC. The words have to be placed between quotes, and they also have to be placed between brackets. The semicolon at the end of the line has nothing to do with the printing action, remember, it's just the signal to the compiler that there is more to come.

The real novelty here is the \n which appears within the quotes and just ahead of **My message**. The \ is the backslash sign, which is on most computer keyboards. If your machine has no backslash, consult your compiler manual to find which key has been used in its place. Don't confuse the backslash with the forward slash that is used in **/*rem*/** lines. The effect of **\n** coming before the text is to force a newline before anything is printed. You could also place another **\n** following the text to cause a new line to be taken after printing. There is a complete set of these backslash instructions, all of which must be included between quotes in a **printf** type of statement. Figure 4.2 shows this set. If, for example, you use **\f** this will carry out a formfeed if sent to the printer, and will (usually) clear the screen if the screen is being used. Note that in this particular line, the quotes enclose both the formatting instruction \n and the text. It's equally possible to separate the two, but the formatting instructions *must* be enclosed by quotes, so it makes sense when the phrase is printed here, as distinct from using a variable name, to use the same set of quotes to enclose both.

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

Any other codes can be put in as numbers in octal code following the backslash.

4.2 The backslash instruction letters that are used as control characters for a printf statement.

In the next line, the formatting characters are placed between quotes, but the numbers are not. The arithmetic **result** is printed out, just as it would be by PRINT 5+3*2 in BASIC. As in BASIC, the multiplication is carried out before the addition, so that the result is 11, not 16. Here again a semicolon has been used to mark the end of the statement, and there's another **\n** used to cause a newline. The novelty this time is the **%d** which follows the **\n**. The % sign is a general way of indicating how you would like an item printed, and when it's followed by a **d**, then the number is printed in denary. If you haven't come across this term before, it means the ordinary scale-of-ten numbers that we use. Once again, there is a whole set of these 'specifiers', and Figure 4.3 shows the complete list. After this line, the main program ends with the curly bracket.

Mark	Meaning
%d	Signed denary number.
%u	Unsigned denary number, range for integer 0 to 65535.
%o	Unsigned octal number.
%x	Unsigned hexadecimal number.
%c	Single character.
%s	String ending with a zero.
%%	Print % sign.

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

4.3 The % specifiers for printf that are used to format numbers, characters and strings. These will decide how something is printed, and sometimes if it is printed at all.

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

Now after typing this program, and saving the text under a suitable filename, compile the program. The text file is called the 'source-code'. Until this source code has been compiled it is just a file of ASCII codes, nothing more. Once compiled, it is object code, closer to machine code and quite different in action. The most important difference from your point of view is that the source code can be read, edited, and is easily understood. The object code has no meaning unless you know about machine code, it is very difficult to edit, and can be recorded only by the compiler or with the aid of a machine code monitor. When you have compiled this program, run the machine code and check that it acts as it should, and that there are no error messages. Having done this, once again this is a good opportunity to make some deliberate mistakes to see how your compiler handles them.

Using a variable

When you use a constant, like 3.1416, in BASIC, you are always advised to assign the value to a variable name. The reason is that this avoids the BASIC interpreter having to convert the ASCII codes for the number into number-variable form each time the number is used. The same is true of C programs, but with the difference that you can either assign to a variable name or use a **#define**. If your compiler is integer-only, then you can't use numbers like 4.1416 , which are called 'floats', numbers which can contain fractions. The whole of C is aimed more at the use of integers, in fact, and the use of floats can sometimes be rather clumsy. To start with, then, we'll look at integer variables only.

The integer declaration uses the reserved name **int**. As we saw in Chapter 2, you can declare that a name will be used for an integer, and then assign a value to the integer. The syntax of declaration is:

```
    int quadro;
```

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

```
    int duo,tri,quadro;
```

if you wanted to declare several names as integers. Note the semicolon to show the end of the statement, the end of that declaration. Once the names are declared, you can make assignments to these names, using integer numbers. Figure 4.4 shows a simple program which makes use of the integer **quin** to mean 5. In this example, the declaration of the integer and its assignment are both straightforward, but the printf line is not. In C, printing is a very different kind of operation as compared to BASIC because of the formatting characters.

```
main()
{
 int quin;
 quin=5; /* assignment */
 printf("\n%d times 3 is %d",quin,quin*3);
}
```

4.4 An example of declaration, assignment, calculation and formatted printing.

The first part of the **printf** statement consists of the words and formatting instruction only. We want two numbers to be printed, both in denary form. In the phrase that is to be used, then, the **%d** is put in each part where a number will be printed in the version we see on the screen. Once the quotes are closed, the numbers are put in, using the same left to right order, and with commas used to separate the numbers. The numbers are **quin**, the integer, and **quin*3**, the result of a calculation. Once again, this **printf** line is a statement, and it has to end with a semicolon. When it prints on the screen, you see the message:

```
5 times 3 is 15
```

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

Constants

The use of a variable for holding a number in C is close enough to the methods of BASIC (so far) to cause you little worry. There is the #define alternative in C, however, for storing items that you might want to use in any part of a program. These constants, as we have seen, have to be defined in a way that is quite different from our definition of variables. The definition of a constant is done at the beginning of a program, before the **main()** portion or (almost) anything else. The syntax is simple enough, **#define** , followed by a space and then the name that you want to use, another space and the value. There *must be no semicolon at the end of a #define line.* Constants can be numbers, single characters or strings, as you please, providing you assign and use them correctly. Take a look, for example, at Figure 4.5.

```
#define mile 1760
main()
{
 int ml;
 ml=3;
 printf("\n%d miles is %d yards",ml,ml*mile);
}
```

4.5 Using #define to place a 'constant' into a program.

The line which 'declares the constant' of **mile** is situated immediately at the start of the program, using **#define mile 1760**. In the main program, the part that lies between the curly brackets, we will use **mile** as meaning the number 1760, the number of yards in a mile (remember them?). This meaning **must** be declared before the program begins. This way, the compiler has allocated memory space for the constant and is ready to use it before the program needs it.

As we have seen, you might have to allocate several constants like this before a program starts. These constants need not all be integer numbers like 1760. They could be float numbers, or letters or phrases, like 'Press any key', and this use of a constant replaces a lot of the purposes for which we use string variables in BASIC. This is important, because C does not have string variables **in the form that we use in BASIC.**

In the example, you can see the **printf** phrase **"\n%d miles is %d yards"** used with the **%d** to specify where the numbers will be printed, as denary numbers. Following the phrase comes the quantities, the integer variable ml and the constant **mile**. In this example, the numbers have been printed as denary numbers, but you can **force** any **printf** action to produce numbers in other forms, such as hex or octal, that you want. You can also decide how much space you want the number to take up. Try a change to the printf line , so that it reads:

```
    printf("\n%-6d miles is %8d yards",ml,ml*mile);
```

and compile and run this one. You will see that the figure '3' appears on the left-hand side of the screen, and the number 5280 is spaced out from the word 'is', taking up 8 character positions. As you may have guessed, the figure 8 along with the 'd' specifies that the number shall be printed taking up 8 character positions, and placed at the right-hand side (right-justified) of this 8-character space. By using a minus sign in front of the number, the space is allocated similarly, but the number is set over to the left (left-justified).

This type of control over number position is called 'fielding', and it's much easier in C than it is in BASIC. If you don't use any numbers along with the 'd' in formatting, a number will simply take up whatever space it needs in the **printf** statement. You can, of course, decide on the number of spaces either ahead of or following the number by putting them in with the spacebar. The fielding method is particularly useful if you want numbers organised in columns, and really comes into its own when floating point numbers are being used, because it allows the decimal points to be lined up, something that can be quite troublesome in BASIC.

I said that a constant did not have to be a number, but could be a character or a string. For items like that, you still use **#define**, with the name that you want to allocate, and the character or string spaced from it. The character or string needs to be surrounded by quotes, as Figure 4.6 shows. If you omit the quotes you will get an error message during compiling when the character or string is used, rather than where it is defined. The message will probably be 'undefined variable', but if you surround the characters with marks other than quotes you can get some quite exotic error messages. Just what you get depends on the compiler that you are using. The other part of the deal is that if you want to print messages in this way, the printf statement needs to be changed. In place of the **%d** that you needed to specify a number in denary form, you need to use %s for a string or %c for a single character. Without these modifiers, **nothing** gets printed!

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

4.6 Using #define for string or character constants.

In the example, %s has been used for both, but we could have used %c for the single character. The use of **#define** in this way allows a lot more than just the occasional number constant or message phrase. With **#define** , you can make your programs much more readable, particularly by definitions such as #define white 0 and #define black 1, which allow you to use words in place of numbers for items such as board games, or allocate values to items, as in #define detected "1000 points".

More variables

We have already made use of an integer variable in a program, and the style is easy enough. There's much more to this type of variable than meets the eye, however. A variable that is declared as, for example, **int num** is an automatic variable. In all varieties of C, you can state this by typing **auto int num**, but if you don't use any word before **int**, then the use of **auto** is assumed, it's the default. Most of the variables that you are likely to use in programs will be auto types, simply because of convenience.

We have noted in Chapter 3 that you would probably want to use auto variables for most purposes, but there are exceptions with some compilers, in which the use of static variables considerably speeds up program execution at the expense of storage space. You might, for example, want to use static variables in games programs to get the highest possible speed. One feature that can be useful right away, however, is initialisation. The way that you can declare and assign in one operation in BASIC, such as A%=5, is very useful. In some small compilers, notably HiSoft on the Amstrad machines, you can't do this with automatic variables, but you can with statics. Take a look at the very brief example in Figure 4.7. This declares a static int **and** makes the assignment of 8 to its value. The **printf** statement then shows that the assignment has been carried out. If you attempt to do this with an auto variable, as by deleting the word static, then you will find whether or not your compiler allows the initialisation of auto variables. The examples in this book have been printed from the latest version of HiSoft C , and so auto variables are always initialised separately.

```
main()
{
 static int squares=8;
 printf("\n%d",squares);
}
```

4.7 Simultaneous declaration and assignment of an integer. Check to find if your compiler permits this for an auto integer.

Getting a value

Suppose that we extend the use of an integer variable to a variable whose value is entered from the keyboard? One of the things that makes C seem very awkward for a former BASIC programmer is the lack of anything like the INPUT statement. Instead, you have to use library functions that are matched to the kind of data you expect to read. One of the standard identifier words for reading an input is **getchar()**, which is a function that reads a single character.

Using this function brings us up against one of the features that newcomers to C find so irritating – the use of characters. The function **getchar()** will get characters from the keyboard, meaning that you can type any character, digit or letter, that you like. If you look up the action of this function in the compiler manual, however, you see it described as **int getchar()** meaning that it gives you (or **returns**) an integer. Whatever you type at this point is accepted as an ASCII code, and this is the integer value that you get. What we are going to type is a number which will have to be assigned to a variable name of x. The value of x, however, will be an ASCII code. For the numbers 1 to 9, this means a code in the range 49 to 57. We can get the number values back from this by subtracting 48, the value of ASCII '0'. Now we can do this in two ways. One way would be the familiar BASIC way, using **x=x-48**. We can, however, also write it as shown in the listing: **x=x-'0'** , meaning that we subtract the ASCII code for zero. This second form is a lot easier to use and understand - for one thing, you don't have to strain your memory for the ASCII codes!

```
main()
{
 int x;
 printf("Please type a number, 0 to 9 \n");
 x=getchar();
 x=x-'0';
 printf("\n Double this is %d",2*x);
}
```

4.8 Using a function to obtain a keyboard ASCII code, and converting to the number form.

Using **getchar()**, as you'll gather, is rather primitive. Though you can type more than one digit, the function works only on the first, which is why the program limits input to the range 0 to 9. There is a function called **atoi()** in the library for converting characters into numbers, but that's for later. There is also, in the library, a routine which corresponds more closely to BASIC's INPUT, but without the facility to mix questions and input like INPUT "Answer: ";A$ in BASIC. The trouble with using these routines right at the start of your conversion to C is that they involve a lot of new ideas, and we can't ever take in too much at one time.

Other variable types

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

```
#define CLS 11
main()
{
 char m1,m2,m3;
 m1=65;m2=66;m3=67;
 putchar(CLS);
 printf("\n%c%c%c",m1,m2,m3);
}
```

4.9 The C equivalent of CHR$ in BASIC, using a character printf statement with ASCII codes.

C, in common with several other languages, has the variable type called **char**, which as we have seen means any character of the computer that is represented by an ASCII code. In some compilers for specific machines this could include the graphics characters as well as the ordinary alphabetical, punctuation and digit characters. Now with this char variable, we can do the actions that you associate with PRINT CHR$( ) in BASIC, and then some more. Take a look at Figure 4.9 for example. The #define line has been used to define the formfeed character as CLS – use whatever code clears the screen for your computer. Three char types are defined, variables m1, m2 and m3. These are assigned with CII codes 65 to 67, and two methods of printing characters are illustrated.

The use of **char** means that the variable consists of one ASCII code, stored in one byte of memory. The putchar(c) routine is called from the library, and it prints one character. This function requires the argument which is shown as 'c' in the library, but for which we can substitute anything we like. In this example, the substitution is to CLS, the screen-clear character. We can, however, use the more versatile **printf** for the other characters, this time with the %c modifier to indicate that a character is to be printed. If your computer and compiler allow the use of graphics codes, then try such codes to replace the alphabetical codes of 65 to 67.

Chapter 5
Data types again

Before we get involved in arithmetic operators, we must do some memory refreshing on data types. The main primary data types are integer (short or long), float (ordinary or double-precision), and character. All of these are **number variables**, because a character is just an ASCII code. Of these, the character is stored in one byte, the short integer in two bytes, and the long integer in four bytes. What is used for floats is very much a matter for your compiler, and some may not accept floats at all. A common arrangement is to use four bytes for an ordinary float, and eight bytes for a double-precision float. Using this amount of storage allows ordinary floats to range from -1.701411E+38 to +1.701411E+38, but with precision to only about six decimal places. This is not really enough even for financial programs working to the nearest penny, because the errors in storing ordinary floats would accumulate until there was a noticeable effect.

The use of double precision allows a much greater precision, to about twelve places of decimals, but at the cost of more storage space and slower handling. For this reason, the C standard provides for all arithmetic on floats to be carried out in double-precision, even if the answer is in single-precision. You need to watch this point if your compiler allows the use of floats, because any floating constants that you assign with #define will normally be treated as double-precision numbers by the compiler. Once more, this is something that you have to check for your own compiler.

Given, then, that all the data types of C are number types, what happens if you mix them? You would hardly think of adding a character to an integer in BASIC, but C is not so fussy, and this has given C rather a bad name among academic programmers. In arithmetic, mixing data types causes the automatic conversion of types until an answer can be obtained.

The first conversions are from char (or short) to integer, and from ordinary float to double. What happens following that depends on what arithmetic is to be performed. The guiding principle is that all numbers will be converted to the longest form that is in use. For example, if the expression contains a double-precision float (after the first conversions), then all other numbers are converted to this form, and the answer is also of this form. If the longest form of number is just an integer, then the answer will be an integer. The important point is that these conversions are automatic, and they are also, incidentally, performed by several varieties of BASIC that use double-precision numbers. The one to get used to is the **char** type, because in BASIC you don't normally think of characters as being single-byte numbers.

Some operations

There aren't many programs that you are likely to write that don't involve the operators of C. The operators are the symbols that control actions on numbers (including characters), and C is rather richer in operators than BASIC. In addition, some of the actions and the order in which they are carried out need rather more thought than you would give to similar things in BASIC. Several operators relate to items (such as pointers) that we haven't met so far, and we'll concentrate for the moment only on operators that will be reasonably familiar.

The four main operators of * / + - are specified just as they are in BASIC, and the only thing to watch for is the effect of division if your compiler happens to be restricted to integers only. The % operator, however, is used to find a remainder. The expression z%x, for example, means ' find the remainder after z has been divided by x' This quantity must always be an integer, and the two variables z and x must also be integers. Even if your compiler is integer-only, the use of the / and % operators allows you to carry out divisions and show the result as a number and a remainder whereas using a float number with the / operator would allow the result to be shown as a number and decimal fraction.

Operator classes

The simple arithmetic operators + - * / and % are all classed as binary operators, meaning that each one of them requires two numbers to act on. Don't be confused by the term binary in this sense, it doesn't mean that you have to write everything in binary coded numbers. This provides us with one very convenient way of classifying number operators as unary (one number needed), binary (two numbers) and ternary (needing three numbers). At this point, then, it looks like a good idea to list just what operators are available in these different classes. Figure 5.1 shows the list of the main unary operator symbols, some of which will not be discussed in detail until later.

*	pointer to variable.
&	address of variable.
-	negate (make negative).
!	change 0 to 1 and 1 to 0.
~	ones complement.
++	increment.
--	decrement.

5.1 The main unary operators of C.

The first two operators * and & are in this class, because when they are used as unary operators, they refer to actions on pointers, and as such belong in Chapter 7. The - sign used as a unary operator simply changes the sign of the quantity to which it is applied. This action is sometimes called the negate.

The ! sign is a form of two-value negate. If you have an expression !x, then the effect of the ! sign is to make the result either 0 or 1. If x=0, then !x=1, if x is any value that is not zero, then !x=0. This can be a very useful operator for test purposes. The ~ operator gives a result that is the ones complement of a binary number. This is rather more specialised unless you are working with binary numbers, and if you are unfamiliar with the ones complement, then consult Appendix A.

```
main()
{
int x,y,w;
x=1;y=0;
printf("\n !x is %d, !y is %d",!x,!y);
x=15;y=-20;
printf("\n ~x is %d ~y is %d",~x,~y);
x=3;
w=++x;y=x++;
printf("\n w is %d, y is %d and x is %d",w,y,x);
}
```

5.2 A program that illustrates the effects of some of the unary operators.

The effects of these unary operators, and the ++ and -- types, are illustrated in Figure 5.2, which lets you see them in action on your own screen. The ! effect is straightforward, and Figure 5.3 explains the effect of the ~ operator on the integer numbers 15 and -20. The next line carries out an action which is quite certainly unfamiliar in BASIC. The assignment **w=++x** means that the value of x is incremented, and it is then assigned to variable w. The next part of the line then assigns the value of x to y, and then increments this value of x again. The printed values of w, y and x then show what has been done. The ++ operator means increment (increase by 1) and the -- sign means decrement (decrease by 1).

Just as important, however, is the point that the **position** of these symbols is important. The printout states that w is 4, y is 4 and x is 5. Using the increment sign **following** the variable name means that the increment action has been carried out after the assignment, not before. If you make the line read:

```
    w=x++;y=++x;
```

then the result is 'w is 3 and y is 5', because w=x++ made w=3, and then x=4, while y=++x made x equal to 5, and then carried out an assignment to y. The use of increment and decrement can be very handy in loops, but you have to think out the order of things carefully. This becomes more difficult when you get to complicated expressions, so it's advisable to start with the easy ones.

```
The number 15 in two-byte binary is
0000 0000 0000 1111, and the ~ operator converts this to 1111  11
11 1111 0000. In unsigned denary this 65520, in signed denary -16
The number -20 in two-byte binary is  1111 1111 1110 1100,
and the ~ operator converts this to 0000 0000 0001 0011. In
denary, signed or unsigned, this is +19.
```

5.3 The effect of the ~ operator on the numbers 15 and -20.

Note that there is no square root among these unary operators. If your compiler permits floats, there should be a square root function in the library. If your compiler does not support floats, you can't use square roots in any case.

Binary operators

The binary operators are the ones that need two numbers to work on, and they are listed in Figure 5.4. Of these, the ordinary arithmetic symbols * / % + and - have been mentioned already. The signs >> and << cause binary-number shifting, and along with &(bit and), ^(bit XOR) and |(bit OR) have been consigned to Appendix B. The remaining operators may look more familiar, but you should not assume that they will work in a familiar way. For example, the comparisons <, >, <= and >= make the usual comparisons of less than, greater than, less or equal and greater or equal, but the result of such comparisons is always 0 or 1, meaning false or true. The == sign means identical to, and the != sign means not equal, and the hard thing for the BASIC-reared programmer to remember is that the meanings of = and ==

must be distinguished. To assign a quantity, = is used, to test for identity, == is used. BASIC uses an equality sign for both purposes, something that is done by no other high level language.

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

Note: There is also one ternary operator (using three numbers to operate on), which is:
?:›Select one or other.

5.4 The binary operators of C. The name is slightly misleading - it means operators that require two numbers to work on, not simply operators that work on binary coded numbers.

The logical operators

The logical operators in BASIC are the words AND and OR that are used to connect two conditions. The problems with their use in BASIC is that they are sometimes used for a rather different purpose also, bitwise operations, so that when you use AND and OR in BASIC you have to know the syntax of your interpreter very well to be absolutely certain what the use of these operators will give you. In C, these two are represented by symbols, and we use the **&&** for AND and the || for OR. Once again, though all keyboards have the & sign, not all have the | sign, and your manual will probably suggest a substitute for this character if it is missing from your keyboard. These two **relational** operators are used in lines such as:

```
        if (n && a) printit;
```

which also illustrate how concise C can be about these things. Translated, this means that if the variables n and a are both non-zero, then run function **printit**. In this example, n and a are **flag** variables, the type that Pascal keeps a special variable type Boolean for. In C, these variables would be of type char or int.

Oddly enough, though an integer variable needs two bytes for storage, and only one bit is really needed for this type of flag, it is often faster to specify int than char. A lot depends on the compiler here, but it's something to look out for. Note how an if test is placed in brackets, and does not require the use of 'then'. The test if (n) by itself would mean 'if n not-zero', and will be true (1) or false (0).

C expects you to enclose in brackets any test whose result will be true or false, so that tests such as an **if** test can be used. This rule is strictly applied, and can cause a lot of syntax errors to appear in your compilations at first. The | | operator is used in a similar way, in a statement such as:

```
        if (name!=' '||x) listit
```

which means that if the character variable **name** is not a blank, or if integer x is not zero, then the function **listit** can run. Note that a character is being tested.

For reasons that will be clearer later, you try to avoid making tests on strings, because there is nothing corresponding to the BASIC test: IF A$=B$ THEN... anywhere in C . To test two strings for equality requires a library function in C , and is something you don't do unless you really need to.

Operator precedence

The listing of operators in Figures 5.1 and 5.4 is in order of precedence in each list. As far as precedence of the list is concerned, the unary operators take precedence over the binary operators. This is the normal arrangement that you would expect from BASIC, and another point that you would expect is that in an expression where operators are of equal precedence, the expression order is left to right. If you have any doubt at all about this, though, you should use brackets. Unlike interpreted BASIC, in which each set of brackets will slow down the evaluation of an expression, the C compiler will deal with this so as to give fast-running code no matter how you dress your variables up in brackets. One thing that you must not take for granted, however, is the order in which the compiler will take the different parts of an expression.

If, for example, you have an expression in which a character is taken from a file in two different parts of the expression, you can't depend on these actions taking place in the order you would expect , which is normally left to right. There is nothing in the specification of C that guarantees this order of evaluation of an expression, and what actually happens depends very much on the compiler, and may not be consistent. If in doubt, it's better to split such an expression into two separate lines.

The ternary operator

C has one ternary operator, meaning an operator that deals with three items. The ternary operator is a conditional which allows two possible results of a test to be carried out in one statement. The syntax is:

```
    expression 1 ? result 1 : result 0
```

and the effect is to test expression 1. If this gives TRUE (non-zero), then the result of the operator is result 1. If the first expression is zero, then the result is result 0. These 'results' can be expressions also, and the principle is that a test can be made to provide two possible results in one compact line.

Other operators

The **sizeof** operator is one that is quite definitely a stranger to BASIC. Its use is to determine how many bytes of storage a variable will need, and as such it comes in very useful later on when we have to make room for variables. Normally, this isn't something that would worry you, but in some C applications, the variable storage size has to be specified. By using **sizeof**, you can place the number into the expression automatically, saving you from having to calculate it for yourself. The **sizeof** operator is one that comes into its own when you are working with structured variables, and we shall meet it in Chapter 10. Another operator, the **cast** operator will be dealt with in more detail in the following Chapter.

The binary number operations

The binary number operators have been left until last in this chapter, because they are not needed so often as the number and character operators. I'll assume here that you have a working knowledge of binary, because if you haven't, then this is no place to start learning. If you haven't, then I suggest you skip what follows until the end of this Chapter. You can learn about binary coding from any of the books on machine code that apply to your computer.

The bitwise NOT operator uses the ~ symbol and is unary. Its effect is to invert each bit of a byte, so that the result is the ones complement of the number. For example, suppose you have a **char** number whose value is 65 denary. In binary, this is 01000001. Performing the NOT action on this gives 10111110, which is in denary 190 **or** -66, depending on how your compiler treats numbers like this. The other problem about this action is finding the ~ sign. It is marked on some keyboards, not on others. On the Amstrad machines, for example, the ~ sign is obtained by pressing CTRL2 (on the CPC6128) or EXTRA- (EXTRA key and dash) on the PCW 8256. There is usually some combination of keys that will give the correct ASCII code of 126 for this symbol.

The bitwise shifts >> and << will shift the digits in a binary quantity by the number of places that is specified in the right-hand part of the expressions. For example, **val<<3** means that the binary number quantity represented by **val** will be shifted three places left. If **val** is the number 01100101, equivalent to 101 denary then the result of this action will be 00101000, 40 denary. If no digits are lost from the left-hand side of a binary number, the result of each place of shift is multiplication by two, but this can go very far adrift if one or more 1's are lost from the left-hand side of the number. The result that you will get, then depends on the number of places shifted, and whether the number is of char size, integer or long integer.

```
Number is 0000 0000 0110 0101

After shifting, this is:
        0000 0011 0010 1000

which is denary 808.
```

5.5 The effect of three left shifts on the number (denary) 101 when written in binary integer form.

The result above was for a single byte, but if an integer number is used, then the result is denary 808, as Figure 5.5 shows. Unless you choose to have both input and output in char form, the default is always integer. The >> sign operates to provide a bitwise shift right, equivalent to division by two, with no remainder.

The bitwise logic operators, &, | and ^ have their usual truth table actions, and Figure 5.6 reminds you of these tables, which apply to each bit. These are binary operators which compare a bit in each byte (or integer pair) with the corresponding bit in another byte or integer. Their uses are mainly in masking and in cryptation actions. For example, bit 7 of a number can be stripped by ANDing with #7F, so that if we take the number 223 and AND with 127 (hex 7F), then the result is 95. In C terms, printf("%d",223&127);
will give the answer of 95. If we want to set the most significant bit, still using a single byte number as an example, then the procedure is to OR with #80, in denary terms 128. For example, the result of:

```
printf("%d",95|128);
```

will be 223 again, since this action resets bit 7 of the number. Figure 5.7 makes this example clearer by showing it in binary terms.

Comparing bit A with B

AND (&) action:

A	B	RESULT
ø	ø	ø
ø	1	ø
1	ø	ø
1	1	1

Summary: The result of A&B to 1 only when both A *and* B are 1.

OR (|) action:

A	B	RESULT
ø	ø	ø
ø	1	1
1	ø	1
1	1	1

Summary: The result of A|B is 0 only when both A *and* B are 0.

XOR (^) action:

A	B	RESULT
ø	ø	ø
ø	1	1
1	ø	1
1	1	ø

Summary: The result of A^B is 1 only when A *and* B are opposites.

5.6 The actions of &, | and ^ shown as truth tables.

These two actions are most likely to be useful when applied to char variables, since this will act on ASCII codes. You can, for example, mask out bit 7 to ensure that no graphics characters are sent to a printer or through a serial link. You can equally easily change between upper-case and lower-case, using bit 5.

```
Number 95 is in binary  0101 1111
Number 128 is           1000 0000
Result of OR is         1101 1111
- and this is 223 in denary.
```

5.7 Explaining the action of an OR with denary 128.

The use of the XOR action for coding characters is well known, but that doesn't ensure that you have heard about it. The principle is that if a byte is XOR'd with another byte, the **key** byte, then the result is a code for the first byte. If this code is then XOR'd with the key again, the original byte is recovered. The encoding action is not confined to one step. If you use a double key, XORing the original byte with one key and then another, the original can be recovered by two XOR actions, using the keys in *either order*. Figure 5.8 shows this in more detail, using a single byte example. Codings of this type, using a word code, can be useful for many applications. The word can be used in a circulating letter sequence, as Figure 5.9 shows, or by using each letter of the word in sequence to XOR each single character of the message (Figure 5.10).

```
Coding-

Byte to be coded   0100 0001
Code key           0100 1101
Result of ^        0000 1100

Decoding-

Code number        0000 1100
Code key           0100 1101
Result of ^        0100 0001
```

5.8 Coding and decoding a byte with the aid of the exclusive-OR action of ^.

The results might not keep the CIA out for very long, but they are not so easy to break that any amateur is likely to get to your message in a moment or so. The only point to watch is that the results of an encryptation do not go outside the ASCII range. For example, if you use:

```
printf("%c",'A'^'F');
```

to print **as a character** the XOR of ASCII 'A' and ASCII 'F', then the result is usually a beep, bell character 07. This is not a printing character, so it makes transmission of the message rather difficult. You may find, then, that more than one XOR is needed on each character just to ensure that the code ends up in the correct range, and this makes the encryptation rather more complicated, since you have to make certain that each step is reversible.

```
Message is:  THE MESSAGE
Key:         KEYKEYKEYKE
```

Results of XOR will be number codes, as for example 'T' ^ 'K'= 31, for each character. As before repeatig the action gives the original character.

5.9 Using a key word, here shown as KEY, to code each letter and space of a message.

```
Message is:    MESSAGE
Key:           KKKKKKK
               EEEEEEE
               YYYYYYY
```

so that, taking the first letter, the ASCII code for ‘M’ is XOR’d with ‘K’, then the result with ‘E’, then the result with ‘Y’ to give the number 26. If this is then XOR’d with the codes for the letters in KEY in any order, the original byte will be obtained for each character.

5.10 Using the characters of a key word in sequence on each character of a message.

Chapter 6
Types and functions

Type conversions

We have already looked at the automatic conversion of types, and it's time now to examine this idea in more detail. The principle is that no action on numbers should be made impossible just because the numbers happen to be of different types, and rather similar principles operate in BASIC, though you are seldom aware of what is going on. It's not difficult to see the sense of the arrangement when you want to add a float like 24.76 to an integer like 21, and get as a result the float number 45.76. In this case of type conversion, it would be ridiculous if we needed any special instructions to carry out the conversion of the int to a float before performing the addition.

Any conversion that makes reasonable sense is performed automatically, and only silly conversions are refused. To give the standard example, you won't be allowed to use floating point numbers as the subscripts of an array. Where the BASIC programmer finds difficulty, however, is in the equally automatic promotion of a char variable (one byte) to an integer (two bytes) when these two types are mixed. A conversion of this type can, for example, take the place of the BASIC STR$ statement. Consider, for example, the action

J%=STR$(D$) in BASIC, where D$ is assigned with "5". The C equivalent of this is simply: **j=d-'0'** , j being assigned with the ASCII code in **d** minus the ASCII code for zero. This assumes that j is an integer variable, and **d** is a char variable. These variable types would, of course, have to be declared earlier in a program.

Complex assignment

Assignment in BASIC amounts to something of the form (LET) X=expression, whether you are dealing with numbers or with strings. Strings are treated very differently by C, as you'll find later, but number variable assignment can be as straightforward as that of BASIC. In addition, though, you can assign a variable in a way that in BASIC would need an expression. For example, in BASIC, you might use:

```
    X=X+2
```

but in C this could be written in the form:

```
    X+=2
```

which looks like a straight assignment. Similarly, the C statement X*=5 means the equivalent of X=X*5, that the new value of X is the old value times 5. All of the arithmetic operators can be used with the equality sign in this way, which makes for compact listing at the expense of readability.

Another action that we now have to examine is type coercion. As the name implies, this means a type change that is not automatic, and which we force by using a statement. The standard C treatment of coercion is not completely satisfactory, because it relies on the way that an expression is written. In HiSoft C for the Amstrad machines, and also in their C for other CP/M machines, the reserved word cast has to be used, so that a statement of the type x=cast (unsigned) variable is used. In an example like this, the variable would be one, such as a pointer, for which no provision is made for automatic conversion. The **cast** statement here converts the variable, usually a pointer, into the unsigned type, and then assigns it to the name x, which must have been declared as an unsigned integer. This coercion has been necessary because there are no rules about converting pointer numbers (which are two-byte numbers like integers in machines that use 8-bit microprocessors) into other types of number variable.

In standard C, however, the **cast** action does not use the word cast, and the example would be typed as:

```
    x=(unsigned) p
```

in which the variable p is of the pointer type. Once you are used to the idea, this method is no harder to follow, but when you are learning C, and even when you have programmed for a little while in this language, the HiSoft idea of using the word **cast** does have a lot of merits.

Whichever way your compiler insists that you perform the coercion action, there will be no doubt about it if you omit it. The usual compiler message is 'Bad type combination', which makes the fault pretty clear. Some compilers may give you a 'Bad conversion' message; but whatever the message, the remedy is to perform the cast operator action.

Functions

We have seen already that a program consists of any **#define** constants, then a **main()** which is followed by the opening curly bracket, some statements, each of which end with a semicolon, and then a closing curly bracket. In the main program, between the curly brackets, you will place all the actions, in sequence, that the program will carry out. Now in short programs, these actions will be simple ones, and they can all be written between the curly brackets of the main program. As your programs become longer, however, you will need to break them into sections, if only for the sake of planning. Just as you can break a BASIC program into a core and a set of subroutines, or functions, you can break a C program into a main block and a number of functions. The similarity between the languages ends there, though. A function in C is called into action by using a name, its identifier, which must be unique to that function.

In addition, values can be passed to a function in ways that are not used by subroutines. The use of a function is therefore something that needs rather more thought than the use of a subroutine. Unlike BASIC, C permits **only** functions, and there is nothing remotely like a subroutine. If you have programmed with defined functions in a BASIC that uses them, you will feel rather more at home with the way that C uses functions.

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

6.1 Calling a function. The example shows the structure of the function.

Take a look for example at the program in Figure 6.1. I have not typed this with indentation because it's such a simple program that you can see the parts of it without the need for indenting sections. This starts in the usual way, and prints a phrase, 'The name is '. It then calls a function **attention**. Now this isn't a function that is built in to the compiler, nor held in the library. It's a function that we have to write for ourselves. The name of the function is **attention**, and it will make use of anything that is enclosed in the brackets. What is in the brackets in this example is a character 'I'. Note that this is 'I' with a single apostrophe, not "I" with quotes. The difference is *very* important. The 'I' is a character with ASCII code of 73, whereas "I" would represent a string - and a string is not a single character as we'll see later. The 'I' is the argument of the function **attention** , the value that has to be passed for the function to use.

Now of course, we can't compile this and run it until there is a function **attention** written. The main() program is ended by the second curly bracket, and following it we type the name of the function, which is **attention(n)**. We can make the letter inside the bracket anything we like, it can be a complete name or a single letter. We then have to declare that we will be using this quantity n, and that it's a character. The curly bracket then opens, there is a statement **putchar(7)** and the character is printed. Now this is represented in the first **printf** line as n. This is a variable name that exists only inside this function, and because n was used in the 'header', as the argument of attention(), it takes the value that was used as an argument when the function was called, which is 'I'. The printf modifier is %c , meaning that the variable which follows will be printed as a single character. The next line is a conventional **printf** - notice that we don't need a modifier because we don't want a new line, and we aren't printing a variable, just a phrase within quotes. The last part of the function is another **putchar(7)**. These putchar functions are standard C and will either be built into your compiler, or part of the library. If they are library functions in your version of C then you will have to consult the compiler manual to find out how to obtain the functions from the library. The **putchar(7)** function will deliver the equivalent of PRINTCHR$(7), the beep sound.

Now when you compile and run, you see the complete phrase appear on the screen. It would, of course, have been just as easy to use the whole phrase in the **printf** line of the main program, but it's always easier to see how something works from a simple example than it is from a difficult one. The important point is that any function you want to use can be called up by using its name, and the brackets are used to pass any arguments to the function. These arguments can be integers, characters or strings, direct as variable names, or as we shall see later, as pointers. When the function itself is written, you write it like another main() type of program. You need to declare any variables that you want to use inside the function, and that includes the name that you have used for the argument. You then carry out whatever actions are needed.

In this example, this has meant calling up other functions, **putchar** and **printf**, which exist in the memory of the machine along with the rest of the compiler. This is typical of the way that we write programs in C - a small **main()** program calls up functions, which in turn call other functions and so on. If you are using C on a CP/M machine, you can use the CP/M **type** instruction to list any demonstration programs that you may have received with your C compiler, and you'll see how very short the **main** program usually is.

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

This *value* is then **temporarily** transferred to variable n for the duration of the function only. The transfer is completely automatic, and is rather different from the methods that you have to use in older varieties of BASIC. The name that is used for n and anything of this kind is a *local* variable, and there is no equivalent to the use of this type of variable in BASIC apart from the BASIC of the BBC Micro and a few other machines. All variables that are declared inside a C function definition are automatically made local. We'll look later at the principle of passing values back from a function, and of using global variables, whose values are retained in all parts of a program.

Header and body

One simple example can't serve to show everything about a C function, and Figure 6.2 ilustrates the more general layout of a function. The first two lines here, preceding the curly bracket, constitute the header of the function which defines what the function is intended to do. The portion between the curly brackets is the body of the function, the statements that carry out the action of the function. The only point that we need to note about the body of the function at present is that it consists of any declaration that has not been made in the header, and of statements which may or may not include a **return** statement. The return statement is closely bound up with what appears in the header, and we'll ignore it for now. It has no connection with the RETURN that you use at the end of a BASIC subroutine.

```
function type  name (argument list)
declare variables passed to function
{
declare other variables
main body of statements
return (value)
}
```

6.2 The general layout of a function. The portion up to the opening curly bracket is known as the <u>header</u> of the function.

The header of a C function has to be designed in a fairly rigidly specified way. The first word in a header is the type of function, meaning the type of quantity that the function will return to **main**. For many functions, either nothing is returned, or an integer is returned, and the default is integer. If nothing appears in this place, then, the function returns an integer. To my mind, though, it's always better to put in the function type, even if its only as a reminder that this will be an integer.

This, incidentally, covers a **char** as well, because of the automatic conversion of char to int. If the function returns anything else, such as a float, double or unsigned, then this **must** be stated in this part of the header, and if you don't want a **char** to be changed to int then you have to put in **char** as the type. Failure to do so will cause an integer to be returned (if possible), or an error message to be delivered.

The next part of the header consists of the function name and its argument list. The name can be any name that you choose, subject to the normal rules about the use of alphabetical and numberical characters, plus the use of the underline character.

Obviously, you have to avoid the names of other existing functions and reserved names, and as we noted earlier, it's advisable to keep clear of names that start with the underline character. Following the name of the function, and enclosed in round brackets, come the arguments, the list of variable names for quantities that will be passed to the function. These, remember, need not be identical to the names that you will use when you call the function from **main**. You can have a function header that start with:

```
    int getit(a,b,c)
```

and call it with **getit(brand,product,price)** providing that the variables are of the correct types and in the correct order. If your function header contains three arguments, two ints and a char, then you cannot expect it to work correctly if you call it using two chars and an int. Each and every variable that has to be passed to the function must be included in this argument list.

Having dealt with the argument list, the argument declarations follow on logically. In this part of the header, each argument must have its type declared. There are no defaults here, if you have put a variable name in the argument list, its type must appear in the declaration, though you may find that your compiler will take type **int** as a default. Whether it does or not, it's better practice to declare *all* argument variables.

Finally, we return to the body of the function. This can include any declarations, particularly of variables that will be used only within the function, but this can include a variable whose value will be the returned value. If for example, you have near the end of the function the line: **return(x)**, then this means that the value of variable x at this point will be returned as the function variable.

If x is an integer, and the function was declared as being of integer type, then this returned value can be printed or assigned. Suppose, for example, that the function name was **calcit**, declared in its header as type int, and with a **return(x)** line . If you called this function by using **printf (calcit);**, then the effect of this call would be to print the value of x at the time when the **return(x)** statement was executed. You **cannot**, however, use a sequence such as:

```
    calcit;
    printf(x)
```

because the value of x is local to the function, it either does not exist or is used for a different value in the main program, or wherever the function was called from. If you want the value that is returned by a function to be used in the calling program, then you have to assign by using a statement for the type:

```
    y=calcit
```

assuming that variable y had been declared as an int variable. Obviously, if the function **calcit** were declared as of type float, then the variable y would also need to be of this same type. You can, incidentally, use **return** without a variable in brackets. Its effect then will be to force the function to end, even if other instructions follow. This is seldom useful, because a function will return automatically when the ending curly bracket is encountered.

Local and Global variables

Unless you have had the fortune to use a BASIC that allows local variables, this is probably one item in C that will cause you a lot of head scratching. As usual in C, the rules are quite straightforward, but you need to run through an example before you really start to understand how they are applied.

In addition, there is the matter of what happens when C tries to be obliging about something that doesn't exist! Figure 6.3 shows a listing which consists of a **main** with three other functions. None of the functions has a value passed to it, because the one variable that is used is declared external to main and before the word **main** appears, making it global to the **main** function and to everything else that is called from main. The important thing to recognise is just how global a global variable is. Any variable that is declared outside a function will be global to all of the functions that are called following the declaration. Obviously, such a variable could not be global in any functions that were used before it was defined. In that sense, then, the most global variable that you can get is one defined before the start of main.

```
int x;
main()
{
x=5;
printf("\n%d",x);
test();
try();
prove();
printf
("\n%d",x);
}
test()
{
printf("\n%d",x);
}
try()
{
int x;
printf("\n%d",x);
}
prove()
{
int x;
x=6;
printf("\n%d",x);
}
```

6.3 An illustration of the rules regarding a global variable.

The listing starts with a declaration, **int x;**, which makes x an integer variable that will be external to the whole program, a global variable. The **main()** is then started, and x is assigned with the value of 5. This value is then printed to prove that it exists. The function **test** is then called. This function does nothing more than print the value of x. The variable name of x has not been declared in this function, nor has any value been assigned. Because of that, the value is still the global value, and that's what is printed by this function.

The next function is **try**. This time, variable x is declared as an integer, but no assignment is made. The value is then printed. When this part runs, what is printed is entirely dependent on what happens to be in the memory, and the trial on my compiler produced the figure of 22513! The global value of x is quite definitely **not** printed. This is because of the declaration. By declaring an **int x;** in the function, we have over-ruled the 'global' variable x in the main program, but since x has not been assigned (initialised) with a value, there's nothing there.

You may feel that 22513 is a pretty substantial nothing, but what has happened is that **int x** has prepared a storage space in the memory for the value of x, and whatever happens to be in that memory space will be printed, garbage though it may be. This is a very important point - unless you initialise a variable value after it is defined, you cannot be certain what the value will be. Some compilers will initialise variables for you, others don't. Unlike BASIC, C can be relied on to provide the most remarkable garbage if you define a variable and don't assign some value to it.

```
(a) 5          (b) 5          (c) 5
    5              10             10
    22513          22782          22524
    6              6              6
    5              10             6
```

6.4 Printouts from the listing of Figure 6.3. (a) The listing as printed, (b) the result of adding x=2*x just before the printf line in function test. (c) The result of deleting the int x step in function prove.

In the function **prove**, the value of x is declared and assigned with 6. The printout, as you would expect, is 6 this time. Once more, the global value of x has been temporarily overruled – the word that is often used is 'hidden', and a new value has been correctly assigned. When **prove** returns to **main** the value of x is printed once again, and once again it is 5, the global value that was originally assigned; no longer in hiding. The assignment of 6 within the function **prove** has had no effect on this final value. Figure 6.4(a) shows the set of numbers that will be printed as the program runs.

A global variable, then, is global only if no other variable of the same name is declared in a function. A function can alter the value of such a global variable, and the altered value will be used from then on . If we add another line in function **test**, just before the printf line, x=x*2; , then the printout will be as shown in Figure 6.4(b). The new value of 10 in the function is printed, and this value of 10 persists outside the function also after the change has been made. Finally, try deleting the **int x**; step in **prove**. This assigns a new value to global variable x, not to local variable x as happened before. The result of this will be that x retains its new value of 6 at the end of the program, as Figure 6.4(c) shows.

To summarise, then, we can pass values to and from functions in two distinct ways. The conventional way is to pass values in the header, and pass **one** value back by way of the **return** statement. We can pass more than one value back by means of pointers, as we shall see later. The less-conventional method, which is the more familiar method for the BASIC programmer, is to use global variables, that can be passed to and from functions, altered or not. This may look more useful, but it's a method that C programmers try to avoid as far as possible. The reason is that it's too easy to lose track of global variables. One or two perhaps may be acceptable, but in a program that might typically contain several hundred functions, some of which will be taken unaltered from the library, global variables are a nuisance.

You should try, therefore, to wean yourself away from the temptations of global variables, so that you can write your C functions in units that you can add to your library and use in all of your programs without modification. All of that leads us to consider what is contained in a C library.

The library

The library is one of the glories of C, because it's the way in which the language can be continually extended and made more useful. The library is a set of functions, written in source code. Its value is that any function in the library can be named in your own program, and taken from disk or cassette to be included in your program. In many versions of C, this has to be done by loading in the whole of the library routines before you compile. Other compilers, notably Hisoft-C, have a very useful variation on library use which allows you to load in only what you need. This means that the disk has to be searched, and this takes a noticeable time, even for a disk.

As with any compiled language, however, the time you spend on this part saves time later when you run the program. A useful compromise method is to keep each useful library routine as a separate file, and to merge it with your own source file as you need it.

When you start working with C there are two really important library files that you need to know about. The first one is called **stdio.h**, and the second is called **stdio.lib**. You will normally need both of them, and you certainly can't use **stdio.lib** unless you already have **stdio.h** in place. The positions in which these are called are also important. The header **stdio.h** must be placed right at the start of a program, before anything else, even before the #define lines. The header is installed by typing **#include stdio.h** . The use of #include with a filename like this causes the compiler to look on the disk for the routines.

The other set, **stdio.lib** goes right at the end of all your program sections, and contains the library routines of which you might want only a few. This is the file which you might dispense with in favour of individual routines merged with your text.

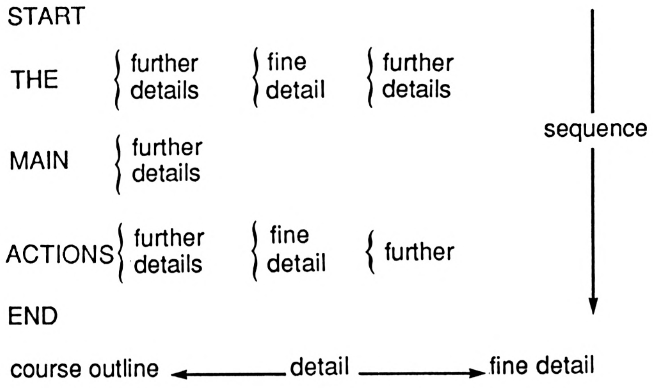
The next step is to decide what to use from the library. You will find as you go along that most of the routines that you need are library routines, particularly as regards string handling. Take a look at Figure 6.5. This defines a string constant as "123fg". The **stdio** routines are used, and the particular routine here is a very useful one, **atoi,** which does the job that VAL does in BASIC. This is to convert a string into an integer number. Only the number characters at the start of the string are converted, just like the VAL action. In this example, the string is a string constant, and the **atoi** action converts it into an integer 123, which is printed in the usual way. There are many more functions which will act on strings, but before we can make really effective use of them, we need to be able to work with pointers, the crowning glory of C. That's for later!

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

6.5 Finding the numeric value of a string, using the atoi function. The steps for including the library may be different for your compiler.

Planning

One of the most important points about functions is how they affect planning of programs. How, for example, do we plan the simple illustration of Figure 6.1? Figure 6.6 illustrates a version of one method that is favoured by many programmers. The left-hand side shows the steps of the main program, with the main action shown as one step. The curly brackets are then used to show where more details are needed. This has been used in the example to show variable names, and also to show what has to be done in the **attention** function.



6.6 A popular planning method outlined. The main objectives are listed in the left-hand side, and each main step is broken into finer detail, using curly brackets to connect the two. The finest detail is on the right-hand side of the diagram.

The important point is that a function is designed in very much the same way as the main program is designed. You can design a function without constantly having to refer to the main program, because the variables that are used within a function need not bear the same names as those used in the main program, the only essential feature is that they should be of the same type. In general, if you define variables for the main program at the start of the program, these variables can also be used in the function. If you define variables inside a function, these variables are 'local', they are used only in the function, and simply don't exist when the function is not running.

Let's look now at a short program, starting with the design steps. This is to be a program which will convert a list of Celsius temperatures into their Fahrenheit equivalents. In case you are using an integer version of C, the whole example will use integers only. The planning, such as it is, is shown in

Figure 6.7. On the left-hand side, the main steps of the program are listed as Start, declare, loop, action and End. The curly brackets now open into more detail. The variables will be called f and c and will be integer numbers. The range and steps will be dealt with by using a loop, which is something new for us, and prepares us for Chapter 8.

```
START

DECLARE     { integers
            { f,c

LOOP        { c=0 to c=100 step 5

            { f= c*9\5+32
ACTION      { print results
            { detail

END
```

6.7 The diagram used to plan a simple Celsius to Fahrenheit program.

The only detail that is included in this list is the range of temperatures, 0 to 100 in steps of 5 Celsius degrees. Fortunately, this range allows exact conversions, so that we don't have to show fractions of a degree, but if we did, then you know how to do this using floats in place of ints. The conversion could be done by a function, but it's so simple that it's hardly worth while, and we simply use the conversion formula.

Now this program illustrates that a FOR loop in C takes a very different form, particularly as regards the STEP portion. Figure 6.8 shows the program which has been drawn up from the plan. As always, looking at a finished program gives you no idea of what the steps were in writing it, so we'll look at the program in the order in which it was written.

The main function starts in the usual way with main(), and then declares the integers f and c. The next step is the loop which carries out the actions of converting and printing the values. Now the first thing to note here is that the for loop line *does not end with a semicolon.* This is because the statement has not ended, we have to specify what will be done in the loop, and that follows, enclosed in curly brackets. The effect of enclosing the two statements in curly brackets is to make this set of lines constitute one single statement. A set like this is often called a 'compound statement', and because it ends with a closing curly bracket, it doesn't need a semicolon. What is inside this set of curly brackets, then, will be carried out on each pass through the loop.

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

6.8 The listing of the program which prints a list of corresponding Celsius and Fahrenheit temperatures. The important feature is the construction of the for loop.

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

```
FOR C=0 TO 100 STEP 5
```

but C writes this as a set of conditions. The first is the assignment c=0, the starting condition for the loop. The next is a test c<=100, meaning c less than 100 or equal to 100. This is the continuing condition. The third is c=c+5 (or c+=5), and this is the stepping condition. If you think that it is all very clear and straightforward, then try omitting the step condition. You'll find that the loop is then endless, and you need to use the BREAK key, or whatever your computer uses for that purpose, to get out of it. Unless you put in a step condition, there won't be any step, and the loop will be endless, unless variable c gets incremented somewhere within the curly brackets following the loop statement.

Unlike BASIC, C provides you with no default step of unity. The other difference from BASIC is the condition c<=100. If you try making this c<100, you'll find the loop goes only as far as 95, because after 95, c is not less than 100. Now try making the middle condition c=100, and see what this does. The effect, another endless loop which prints '100 C is 212 F', is most unexpected if you are still thinking in BASIC terms. The reason is that *each latter part of the loop statement should be a condition* or a test . The c=0 part is a starting assignment. The c<=100 is a test for continuing, but the loop does not end until this condition is FALSE. If you type c=100, this is not a test, but an assignment, and it's not any kind of condition. If you put in a condition which makes the loop impossible, the result is an endless loop. With c=0 assigned to start with, and c=100 used in place of c==100, the loop is endless because there is no ending condition.

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

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

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

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

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

More functions

It's time to take another look at a function action, one which is simple but rewarding to consider. This time, as the listing of Figure 6.9 shows, the function is called (or *invoked*) as part of a **printf** statement. The other important point which this program illustrates is the **return** way in which a value can be passed back from a function. The loop makes use of numbers from 0 to 10, and the incrementing is taken care of by using x++ as the third term in the **for** statement.

```
main()
{
int x;
for (x=0;x<=10;x++)
printf("\n%d cubed is %d",x,cube(x));
}
cube(a)
int a;
{
return(a*a*a);
}
```

6.9 A function which returns a value by way of the return statement.

Since there is only one statement in the loop, it can follow the **for** part, and be terminated with a semicolon which now marks the end of the loop. If you put the semicolon after the **for** statement you'll get a loop, but with nothing in it! The **printf** statement prints the value of x, and also the value of **cube(x)**. Now **cube(x)** is a function, and since we have not stated otherwise, it can return a value which is an integer. The statement **return(a*a*a)**; provides the value that is to be returned. Remember that a is local to the function, it has no value in the main program. The value of **x** is passed to **a** when the function starts, and the function therefore acquires the value of **x*x*x** at the return step- there is no change to the value of **x.**

Once again, you are not forced to pass values to and from a function in this way. You could have declared and used **x** as a global variable in the way that as already been described. You might feel more familiar with such a method, but by using it, you shut yourself off from the ability to make the cube routine one of your library routines, able to be used in any of your programs. This is the whole point of functions in C as compared to subroutines in BASIC.

A BASIC subroutine, because of its line numbers and its variables, will almost always have to be changed if it is used in a program other than the one for which it was originally written. A C function is, by contrast, like a diamond, for ever. Providing that the function uses local variables, with values passed in through the header, and passes values out by using return or by use of pointers, then it is a universal function, and you can place it into any program without having to edit the function to change variable names or anything of the sort. Benefits like this are not to be thrown away just because you were brought up on global variables!

The comma operator

Since this book is aimed at the beginner, it's doubtful if the comma operator should feature here. It's not supported by my compiler, and it might not be by yours. In brief, and with simplification, the comma allows you to do several things at once, and the usual application is in a for loop. Instead of having a loop which includes:

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

you can put the decrementing of n in as part of the loop structure, using the comma, as:

```
    for (j=0;j<=100;j++,n--).
```

Where the comma is used in this way, the expressions are executed in left-to-right order. In this example, there's nothing returned from the expressions, but if there is, then it's only from the rightmost expression. Definitely not for beginners!

Chapter 7
Pointers

A pointer is a type of number variable, and the reason for its name is that it 'points' to where something is stored. For example, suppose that you have the character 'C' stored in the memory of your computer. What this actually means is that one of the bytes of memory is storing the number which is the ASCII code for 'C', the number 67 denary.

Now memory for a computer is organised so that each unit byte is numbered, and we might know that the number of the byte which held our 'C' was 24236. This number of 24236, then, is the number which is the pointer for the storage of 'C'. We could, if we liked, store the number 24236 somewhere so as to make it possible for the computer to find where 'C' was stored, and this is precisely what the action of a pointer is. It would be rather a waste to store a pointer for every single character, but we don't need to. All we need to do is to store a pointer to the **start** of any variable. Once we know where the start is, we can locate it and read the required number of bytes of data. This is something that is used a lot in assembly language programming, but seldom occurs in BASIC. A few machines, notably the MSX machines and the Commodore 128, have a BASIC function called VARPTR or POINTER which comes back with a number that tells you where abouts in the memory a variable is stored.

The Amstrad machines achieve the same effect with the use of @ preceding a variable name. In BASIC, however, there is little use for this action except in sorting string arrays, and not many programmers make use of it, or are even aware of it. In C, however, pointers are a way of storing variables and getting access to them. This is not just a useful feature of C, it's something that is central to the way that the language is constructed. Without pointers, you simply don't get very far with C.

Take a look at a simple example just to get the taste of all this. Figure 7.1 is a program which declares two variables of type **char**. One of these variables is **ccr**, which is a straightforward variable name. The other, however, is referred to as *p. Now the asterisk, in this context means 'contents of'. What it implies is that **p** is an address in the memory, and a character can be held at that address. The program assigns the variable ccr with 'C', a single ASCII code, and then assigns the pointer by using p=&ccr. The & operator, used in this context, means 'address of'. The effect, then, is to store the character 'C' as variable name ccr and make p point to this address. If we assign ccr with something else, then p will point to whatever that happens to be.

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

7.1 A character and its pointer. The last line allows you to find what memory location is being used for the pointer on your machine.

The program then prints out the character, in the form ***p** , and the pointer value itself, which when I ran it on my machine gave 41967. You can expect different numbers to appear on different machines, because where your computer stores its variables is a matter between it and its maker. By using %u with printf, incidentally, we ensure that the number which is printed is positive.

A pointer in C is a variable quantity which is the address of another variable. What makes the pointer valuable is that if the pointer is declared, the compiler can work with the pointer rather than with what it points to. For example, if the pointer to a real number variable is known, and is p, then the real number can be operated on without necessarily referring to its variable name. When 'C' deals with a variable name, in fact, that name is only an indication, a convenient way of referring to the address of a variable, which is the pointer to the variable.

A pointer reserves space for a declared type of variable, what you then put into the space later is your own business, provided that it's the correct variable type. In addition, the pointer is a number (unsigned), but what it points to can be any type of variable, simple (such as another integer) or structured (like a record which consists of a number of different types). We can then juggle with the pointers rather than with the variables themselves.

```
main()
{
 int x,y;
 x=10;
 square(x,&y);
 printf("\n%d %d",x,y);
}
square(x,p)
int *p;
{
 *p=x*x;
}
```

7.2 Passing a pointer to an integer so that a function can use the pointer and so alter the variable.

All of this sounds rather academic, so take a look at an example which reveals a little of what all this is about. In Figure 7.2, two integers x and y are declared in the main program. The variable x is assigned with a value, but y is not. The function **square(x,&y)** is then called. The quantities that have been passed here are the value of variable x, and the pointer to y.

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

Now this is strong stuff - the quantity that has been stored in a variable y has been changed without the need to have a line y=x*x, or even a direct reference to y , all that has been done is to pass &y, the pointer to y, to the function. This is the way in which a function can return more than one value to a main program, and it's a method that is very extensively used in the C library functions. In order to make any substantial use of the library functions, we have to cope with this idea of using pointers. At the moment, one problem that has been hanging over us is how to enter numbers, so this seems a good time to introduce one way, the **scanf** function.

Now as it happens, **scanf** is not the easiest of functions to use, and a lot of programmers avoid it like the plague . The principles are reasonably straightforward, though, and it's principles that we want to look at. Function **scanf** is set out very much like **printf** , with a control section, and a list of the variables that you want to input. So far, it sounds just like good old BASIC INPUT A,B,C. The important difference is that

scanf requires pointers to variables, not just variable names by themselves. The other thing, the one that causes a lot of frustration, is that **scanf** works to strict and rather old-fashioned rules, and can do the most amazing things if you don't understand the rules, or forget how strictly they are applied.

```
main()
{
int j,k;
for(j=0;j<=5;j++)
 {
 printf("\n Small number please- ");
 scanf("%d%c",&k);
 k=k*k*k;
 printf("\n cube is %d",k);
 }
}
```

7.3 Using a pointer with scanf.

Figure 7.3 illustrates a **scanf** action, and gives you a taste of its use of pointers and one of its peculiarities. The main program sets up a loop which will run from 0 to 5, 6 passes through the loop. In each loop, we want to print a brief message, input a number, calculate its cube, and then print that value. The input of the number is the action that is assigned to **scanf**, and the syntax for this particular example is:

```
scanf("%d%c",&k);
```

which at first sight, looks rather baffling. Run it, and check that it does as it ought to, remembering that all the arithmetic is integer, so that squaring large numbers would give very peculiar results. On the whole, it does as you might expect, though you'll notice that there has been an extra blank line. This is because of the use of (RETURN) to terminate the **scanf** line. You can miss out the **\n** portion of the second **printf** statement if you want to close it all up.

So far, so good, but what's the **%c** for in the **scanf** statement? You'll see if you try the program with this removed. The first number is accepted, but following that one, the loop cycles round without waiting for you to enter anything else! The reason is that you used the RETURN key after typing the number. Without the %c in the **scanf** specification, the RETURN character is stored and used each time **scanf** comes round. Since you don't have time to enter a number, nothing else gets entered. Unless you are working with a loop, this action is of no importance, and a lot of books on C don't even mention it. By adding the %c into the 'specifier' part of **scanf**, you allow for the RETURN character, and the loop works correctly. There is another way of getting round the problem, which is to leave a space ahead of the **%d** specifier. The use of %c is a lot more visible, and for that reason, I prefer it.

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

From a brief description of pointers and of **scanf**, it's only too easy to run away with the impression that this is something that's not terribly difficult. A lot of newcomers to C read the 'standard' textbooks, feel that they know it all, and then sit down to write a ten line program which to their disgust never seems to run correctly. This isn't due to an inherent difficulty in the language, it's mainly because it's so different from the more common languages, and because it allows you so much more latitude. You can do things in C that in BASIC would cause an instant error report. C allows you to do what you want, silly or not, and delivers the results, and that's when you find out whether you have blundered or not. We'll take a look now at some ways in which you can go wrong, both with pointers and with **scanf**, because these two are so closely tied together.

To start with, look at the two listings in Figure 7.4. These look as if they are both intended to do the same thing, to allow you to input a character and then print it on the screen. The difference is in the way that the character is defined and the pointer used. In Figure 7.4(a), the declaration is the conventional **char p**, meaning that variable name **p** represents a character. In the **scanf** line we use **&p**, the address of (or pointer to) p, as **scanf** requires. When we print the character, we print it as p, because the character has been put into the address of p by the **scanf** function. A risky and highly inadvisable alternative method is shown in Figure 7.4(b). This time, the declaration is **char *p**, meaning that p is a pointer to a character. Unlike the p in the first version, which was a character, this p is an address, a pointer. The trouble is that your compiler may simply pick an address at random, which is no problem if you change the address assigned to **p**, but can be big trouble if you store anything in this address.

(a)

```
main()
{
char p;
printf("\nType a character\n");
scanf("%c",&p);
printf("\n It was %c",p);
}
```

(b)

```
main()
{
char *p;
printf("\nType a character\n");
scanf("%c",p);
printf("\n It was %c",*p);
}
```

7.4 Character input and printing. (a) Using the correct method of declaring a character variable. (b) An incorrect and risky listing in which a pointer is declared, but nothing is done to set the pointer to a valid address. Using this type of declaration can lead to memory corruption.

As it happened, my compiler allowed this example to run, but yours might not. In the **scanf** line, then, we can use p directly, since **scanf** needs a pointer and this is one. In the **printf** line, we need to use *p, meaning the character that is stored in pointer address p. Notice that in this version, there is **no variable of type char**, only a pointer, or address, for storing a character. The danger in this form is that unless some assigning action is used that makes the pointer point to some established address, pointer **p** will usually point only to garbage.

These two examples are important, because one of the things that haunts newcomers to C is confusion between a variable and its pointer. This is particularly a problem when **scanf** is being used, because **scanf** forces you to use pointers. One very common mistake is to use & and * the wrong way round, another is to confuse the pointer and what it points to. Just to make things more awkward, these confusions don't normally cause error messages to appear. Even more confusing is the fact that some mistaken listings may work! If a **scanf** line is followed by a **printf** line, as it is in this example, you can usually see when something has gone wrong. If, for example, you typed the incorrect version of Figure 7.4(c), then it compiles and runs, but what you type is not what's printed! The reason is not exactly straightforward, but it can be explained if you make some intermediate printings by putting in a line **printf("\n%d %d %d\n",p,*p,&p)**. When this line runs, you'll see that ***p** is the ASCII code for the character that appears on the screen, and the quantities p and &p are (different) numbers.

At the start, p has been declared as a pointer to a character. In the **scanf** line, what is supplied is &p, the address of the pointer itself. The character is therefore p, whether you like it or not. The trouble here is that p has been declared as a pointer, which is a two-byte number, so that when you print p as a denary number, it has a second byte which makes it look quite different from the character. In addition, this two-byte number will point to somewhere in the memory, so that *p will give some character- but certainly not the one that you typed. On my machine, for example, when the 'A' key was pressed, the printout gave 22081 0 -23568. The -23568 is the denary version of the address of a pointer to p. The value of p is 22081, which when you split it into high-byte and low-byte turns out to have a low-byte of 65, the ASCII code for A. The high-byte is whatever happens to be stored in the next piece of memory along! It's not surprising, then, that if this whole number is used as a pointer, then what is points to is garbage! C allows you to place and use any garbage that you like in the memory. You get no warnings, no error messages. If the garbage that you place happens to zonk out the compiler or the operating system, that just too bad.

This means that if you program in a slap-happy way, you'll run into deep trouble at some stage. The worst type of trouble is that a program seems to work perfectly well, but the whole system crashes after some time. It's rather like a game of roulette, but in this case you are waiting for the memory corruption to hit something important. This is why it's so important to write C programs from tried and trusty functions.

```
main()
{
char *p;
printf("\nType a character\n");
scanf("%c",&p);
printf("\n It was %c",*p);
}
```

7.4(c) This is the incorrect version of the program in 7.4 (a). The character is being assigned to the pointer not to what it is pointing to.

After all that, it's time we did some more with **scanf**.The examples of Figure 7.4 (a) and (b) will accept any characters, including things like the space, tab and RETURN. These characters are called 'white space', because attempting to print them in the printf statement simply produces a white space character. The **scanf** statement in this case reads one character and one only. When we come to deal with arrays, we shall see how such white space can be skipped over. Before we get to arrays, which need some experience with pointers to handle to best advantage, we'll look at some other ways that **scanf** can be used, and at some other aspects of pointers.

To start with, **scanf** allows a wide range of 'specifiers' for its inputs, and Figure 7.5 shows the standard list. This does not, however, reveal the splendid quirks of **scanf**. To start with, make the scanf line in Figure 7.4(a) read:

```
scanf("h%c",&p)
```

(not "%h%c", which is different)

and try this version. If you simply type a letter **which is not h**, then what is printed out is garbage. If you type a two-letter combination, with h as the first letter, then the second letter will be correctly printed. In this example, the h is part of the specifier string, a kind of key which unlocks the action for you.

scanf is controlled by a string, like printf, but with the string used to control input format. If the string contains 'whitespace' characters like \n , these will be ignored except when a character input is specified. The main specifiers are used following the % sign, and these should match the next character that is not a whitespace character. These specifiers may be preceded by a field number, positive for right justification, negative for left justification. If the compiler supports long integers, the letter l can be added at the end of the field number to indicate a long integer. The control characters are:

d	to indicate a signed denary number
o	unsigned octal.
x	unsigned hex.
h	short integer.
c	single character.
s	string.
f	float or double in ordinary denary or exponent-mantissa form.

7.5 The <u>scanf</u> specifiers, and how they are used.

Using %h rather than plain h gives no output at all with my compiler because this form of specifier is for a short integer, and we have specified a **char**.

Now try using a specifier "%*c". This will produce only garbage, because the * symbol in a **scanf** line causes the assignment to be skipped – you press a key, but the ASCII character is not placed into the pointer address. Whatever happens to be in that address, whether by a previous assignment or by chance (as in this example) will then be used. I did say that the rules were strict!

Numbers again

One particular advantage of using **scanf** and **printf** together is the way that they can both format numbers. Suppose, for example, you wanted to knock up a quick program for inputting a denary number and getting the hexadecimal or octal equivalent. A lot of BASIC versions offer a hex converter nowadays, but octal is rather an oddity.

```
main()
{
int x;
printf("\n Type a number...");
scanf("%5d",&x);
printf("Hex is %x",x);
}
```

7.6 A simple denary-hex converter using the conversion facilities of scanf and printf.

Figure 7.6 shows a conversion to hex, using **scanf** and **printf**. In the **scanf** line, the specifier "%5d" will allow a denary number of up to five figures to be used. If you use more than five figures, only the first five will be accepted. The output uses the "%x" specifier, which is the code for hexadecimal conversion. The conversion isn't foolproof, because with x declared as an integer, its maximum value is 65536 denary, #FFFF in hex. If you enter a number greater than 65536, then, the hex conversion lacks a leading digit. If this might be a problem, your compiler may allow you to use a long integer type in the declaration, otherwise some smart trapping will be needed between the **scanf** line and the **printf** line. Now if you want to make this into an octal converter, all you have to do is to alter the specifier in the **printf** line so that it reads %o (oh, not zero), and also alter the message to read octal in place of hex.

```
main()
{
int x,j,k;
k=0;
for (j=1;j<=5;j++)
{
 printf("\n Number please (two digits max)");
 printf("\n%d ",scanf(" %2d",&x));
 k+=x;
}
printf("\n Total is %d",k);
}
```

7.7 A number-totalling program, showing that scanf is a function that returns a number.

Another aspect of **scanf** is illustrated in Figure 7.7. This program is a simple one for summing five numbers entered at the keyboard. To restrict the total to an integer, each number can consist of no more than two digits, and the sum is made by using the line **k+=x**, the shorthand version of k=k+x. The novelty in this program, however, is the use of **scanf** within a **printf** statement. This is not so odd as it might seem. What you must remember is that scanf is a function, and any function in C must return *something*.

As it happens, **scanf** is an integer function, so that it has an integer value, which turns out to be 1. By using a printf line with scanf embedded in it, we print this figure of 1 each time which has nothing to do with the action of scanf. In this example, it's simply a demonstration, but the fact that **scanf** returns a number in this way could be useful if we wanted to count up how many times a **scanf** statement had run, for example. Figure 7.8 shows this in action. The counting is pointless, of course, where a **for** loop is being used, but the principle is handy, because as we shall see in the following Chapter, there are other loops which don't keep an automatic count. In the example, the **scanf** statement is included as part of a piece of arithmetic. It's this ability to put functions in as parts of statements which provides a very sharp contrast between C and BASIC, and it's something that takes a lot of getting used to.

```
main()
{
int x,j,k,y;
k=y=0;
for (j=1;j<=5;j++)
 {
  printf("\n Number please (two digits max)");
  y+=scanf(" %2d",&x);
  k+=x;
 }
printf("\n Total is %d in %d entries",k,y);
}
```

7.8 Using the number returned by scanf to count the number of entries in a totalling program.

The way that pointers are allocated can also give some bother. In Figure 7.4(b), the declaration **char *p** was used as the first step in a main program. What value this pointer p will take depends a lot on your compiler, and on most compilers will simply be **any garbage number**. You might get away with it; on the other hand it's just as likely that this pointer address will be somewhere in the compiler or the operating system. You can never make use of declaration of the type *p,*x etc. unless you do something to assign the pointer values. This could be by having a variable previously declared, as in **int y,*p;p=&y**, which avoids problems by making pointer p equal to the pointer of an assigned variable y. At the instant when *p is declared, however, its value is taken almost at random. The other safe way in which declaration of *a,*b etc. can be used is within a function. If these quantities have values passed to them by the header of the function, then they will be correctly assigned. For example, the function header:

```
safeone(x,y)
int *x,*y;
```

presents no problems because the values of x and y are being passed through the header and will, we hope, be sensible values for the pointers that exist in the main program. This, once again, is a point (sorry!) that is unexplained in most books, and you are left to puzzle out for yourself why an **int *x,*y** works in one place but seems to cause havoc in another. Once again, this is the kind of thing that gives C a bad name.

```
main()
{
int x,y;
printf("\n Input two numbers");
scanf(" %d %d",&x,&y);
printf("\n x is %d and y is %d",x,y);
exchange(&x,&y);
printf("\n now x is %d and y is %d",x,y);
}
exchange(a,b)
int *a,*b;
{
 int c;
 printf("\n%u %u %u",a,b,c);
 c=*a;*a=*b;*b=c;
 }
```

7.9 An exchange action that works by exchanging pointers.

As an example of how correctly-used pointers can be useful, take a look at the listing in Figure 7.9. In this example, integers x and y are declared, and you are asked to input two numbers. The input uses a **scanf** step - note the space ahead of each % sign to allow for 'white space'. This is particularly important for the second specifier, because there has to be a white space separator between the numbers when they are entered. You can enter the numbers by typing one, then RETURN, then the other (and RETURN), or you can type a number, then a space, then the other number, then RETURN.

As usual, **scanf** requires the pointers to the integers, which are **&x** and **&y**. These will be sensible values, because the integers **x** and **y** have been declared. Declaration of the integers prepares space in memory, and that space is addressed by the values of &x and &y. These pointer values are then passed to a function **exchange**, and the result will be to interchange the values of x and y, as the last printf line reveals. The exchange function has in its header the quantities a and b which will contain the pointer addresses for x and y. These numbers a and b have to be declared in the pointer header and since they are pointers to integers in this example, that's how they are declared.

In the body of the function, an integer **c** is declared. This could not be declared earlier, because it's not part of the header, just a local variable. The next line exchanges values. Variable **c** is assigned temporarily with the integer to which **a** points, in other words, **x**. The statement ***a=*b** means that what pointer **a** points to is changed so as to be identical to what pointer **b** points to, which is the value of **y**. Next, what pointer **b** points to is arranged to take the value of **c**, the old value of **x** All of this leaves pointer **a** pointing at **y** and pointer **b** pointing at **x**. These numbers, however, are the pointer addresses for **x** and **y** respectively that were passed into the function. Since **&x** now contains **y** and **&y** now contains **x**, the values are swapped when you print them out. Note in particular the two different types of quantities that we are working with here.

Quantities **a** and **b** are pointers to integers, and quantities ***a, *b** and **c** are integers. We can swap values of identical types, so that we can swap among ***a, *b** and **c,** and we can also swap between **a** and **b,** but we can't exchange with different types, such as ***a** and **b.** This may seem odd when all the quantities concerned are stored as two-byte integers, but the division is important - we can make enough mistakes with 'C' without any extra freedom!

The important feature of this use of pointers is that it enables you to change variable values by means of a function . This is the only exception to the rule that every variable declared in a function is local so that anything altered in a function is local also. The use of pointers allows functions to pass back as many values as we need, rather than the limited ability of **return**, which can pass only one value back. This is the type of use that makes pointers so valuable, and it accounts for the number of library functions in a C library which start with a header-full of pointer declarations. It's also the reason for **scanf** using pointer variables, because the purpose of **scanf** is to pass values into the main program, or into another function, and to do so it simply has to make use of pointers.

Chapter 8
Loops

The FOR loop

We have met the **for** type of loop already, and in this section, we shall simply summarise its syntax and action fairly briefly, then add some points that were not considered earlier. To start with, the keyword **for** is followed by a set of three conditions. These conditions can be independent of each other, though for the most common applications of the **for** loop, the conditions all concern the same variable, a counter variable. The conditions are enclosed in brackets, and each condition is separated from its neighbours by semicolons, though see the note on the comma operator at the end of Chapter 6. The first condition is an initialisation, and for the conventional counting loop it gives the initial value of the loop variable when the loop starts. This loop variable will have been declared, and will normally be an integer. The next condition is the 'keep looping' condition, of the form of x<=5. As we have seen, it's important to form this condition correctly because looping will continue for as long as this condition is true. The final condition is the stepping condition, usually the increment or decrement of the variable. Normally we use something like x++, a simple increment, but we could use steps like x*=2, meaning that x will be doubled at each step. The arrangement inside the brackets is therefore of an initialisation, a test, and an action. These need not be on the same variable, and need not be connected, though for most of the loops that we use the connection is that a single counter variable is used.

The for statement is one which **does not normally end with a semicolon.** There are two ways of marking the extent of the loop. One is to omit any semicolon until the end of the loop, and this can be used if there is only one statement line in the loop. The other method is to open a curly bracket in the next line, so that every statement between this and a closing curly bracket is executed as part of the loop. This latter method is to be preferred if the loop is being used for anything more extensive than a **printf** line.

As usual, C has surprises for the BASIC programmer. Take a look at the listing of Figure 8.1, which shows a character variable being used in a loop. The surprise here is the character being used as a counter in a loop – it's no surprise if you have programmed in Pascal, but it's entirely foreign to BASIC. Once more, it's an example of the way in which C treats characters as ASCII codes, which after all is what they are. Note in particular how the character values are assigned, as 'a' and 'z'. This is something you have to be careful about, because if you use "a" or "z", things will go drastically wrong. The symbol 'a' means the character a , but "a" means two characters, the code for a followed by a zero. There will be more of that when we get to strings in Chapter 9.

```
main()
{
char a;
for (a='a'; a<='z';a++)
printf("%c ",a);
}
```

8.1 Using a character variable as the controlling number in a for loop.

While we have a loop operating, we can take the chance to make some changes which will illustrate a few more points about how C uses loops. In particular, C has statements that allow you to skip passes through the loop, or to break out of the loop, without giving the computer's operating system apoplexy. BASIC is not nearly so well organised in this respect. Take a look at Figure 8.2. The loop has been marked out with curly brackets this time, and to make it more readable, I have indented this portion. In the brackets, there is a new statement:

```
if(a=='n') continue;
```

whose syntax and effect is rather interesting. The effect is to omit the letter 'n' from the printout! In this sense, **continue** means continue the loop from the start again, incrementing the variable, but it implies that anything that follows the **continue** statement will be left undone. In this example, that's printing the letter 'n'. The syntax of **continue** will normally use an **if** test, and this is made in the way that is shown. The if keyword is followed by a test **which must be enclosed within brackets.** The result of this test must be TRUE or FALSE, and another important point is the testing for equality.

```
main()
{
char a;
for (a='a'; a<='z';a++)
 {
 if (a=='n') continue;
 printf("%c ",a);
 }
}
```

8.2 Illustrating the use of continue to skip over part of a loop and return for the next pass.

Unlike BASIC, this uses two equality signs, or if you are testing for inequality, the != sign. You can also, of course, use the > and < signs in this context. When whatever is in this bracket is TRUE, then the continue action will be taken. What do you think would be the effect of using the condition: **if (a%2==0)** in the continue line? Try it, if you are in any doubt. The **continue** statement is a way of excepting certain items (even-numbers, short names, one particular name) from being treated by the action of a loop. You can choose your position for the **continue**, too, because you might want to do some of the loop actions before you skipped the rest.

The other loop modifier is **break**. This, as you might expect, allows you to break out of the loop altogether as the result of some test. In the example of Figure 8.3, it's when the value of a reaches letter 'v', so that the loop prints out only as far as letter 'u'. In this example, of course, it makes little sense to do this, because we could just as easily have put this in as the ending condition at the start of the loop. You might, however, want to test for another condition, such as something non-numerical, in this way.

```
main()
{
char a;
for (a='a'; a<='z';a++)
 {
 if (a=='v') break;
 printf("%c ",a);
 }
}
```

8.3 Showing the action of break in a test to end a loop. After the break action, the next statement that will be run is the one following the end of the loop.

If your loop was, for example, reading a list of 100 names, you might possibly want to stop when you found McTavish, knowing that this name could be anywhere in the list. The **continue** and **break** actions of **C** avoid the messy and unpredictable effects of using GOTO's in BASIC loops, and the ability to use this type of loop with a character counter opens up opportunities that don't exist in BASIC.

You can, incidentally, use a **goto** if you are absolutely determined to. Since C uses no line numbers, you have to mark each place that you need to jump to with a label word, followed by a colon. This can be in a line of its own preceding the piece of routine that you want to jump to, or at the start of such a line, like **mark:x=5;**. The GOTO step then consists of a statement like **goto mark**;. Because of the provision of another two loop types, along with **continue** and **break**, the **goto** action is hardly ever required.

While and Do loops

The **for** type of loop can be used when some type of number is involved, whether this is a counter number as we use in the FOR...NEXT loop of BASIC, or the ASCII code for a character as we have illustrated. Unlike many varieties of BASIC, C is rich in ways of creating loops and there are two other loop types in C that allow loops to be controlled by other types of tests.

The difference between the two is important - the **while** type of loop makes a test at the start of the loop. If this test works out to be TRUE, then the loop runs, and will continue to run each time the test gives the result TRUE. Since the test is at the start of the loop, it's possible that the loop might never run, if the test happened to give FALSE first time round. The other type, the **do** loop, makes its TRUE/FALSE test at the end of the loop. Using this kind of loop, you can be certain that the loop actions must run at least once, and will run once even if the test turns out to yield a FALSE result.

The WHILE loop

The syntax of the while loop is simple and will already be fairly familiar to users of machines such as the Amstrad that use this type of loop in their BASIC. The general form of a while loop is:

```
while (condition)
{
statements;
}
```

using the usual method of placing the statement within the loop in curly brackets. As before, however, if only one statement has to be executed, it can be followed by a semicolon to mark the end of the loop. Also as before, the condition must be something that can be evaluated as FALSE or TRUE. The loop will continue for as long as this condition returns TRUE.

Figure 8.4 illustrates a very simple type of **while** loop in action. One point to watch is that since a **while** loop makes its test at the start of the loop, whatever is being tested will have to possess a suitable value at this point, otherwise the loop will not run at all. In this example, this is done by assigning the value of 'a' to the character variable that is being used. The ending condition is for the character to be ASCII 32, the spacebar. The reason for choosing the spacebar is that the RETURN key is needed for entering the value, and it's by no means easy to arrange things so that pressing the spacebar by itelf will operate the test. This is because **scanf** has been used as the input routine, and no space has been put in front of the **%c** specifier. Putting a space into the specifier prevents any number from being returned by pressing RETURN, and does not return a number for any other white space character either. By omitting the space, the spacebar character can be detected and used.

```
main()
{
char k;
k='a';
while (k!=32)
 {
 scanf("%c",&k);
 printf("%c",k);
 }
}
```

8.4 A simple while loop. You must make certain that the quantity that is tested at the start of the loop has a suitable value on the first pass through the loop. This has been done by an assignment in this example.

At this point, we can dispense with **scanf** for a time and look at the use of some of the functions that will operate character by character. One such is **getchar** which as the name suggests will get a character from the keyboard. The action is that characters are obtained from the keyboard (the standard input) into a buffer until the RETURN key is pressed, but even that description does not quite prepare you for the action of the loop in Figure 8.5. This is a typical condensed action line that you so often find in C programs, with a lot of activity poured into a few instructions. Because this is so unlike the structure of BASIC, it will bear a long hard look, and we have to start inside the inner brackets at the statement **k=getchar().** This is the normal way of using the **getchar** function, and its effect will be to get a character from the keyboard and assign this character to variable **k**, which has been declared as a character variable.

```
main()
{
char k;
while((k=getchar())!=9)putchar(k);
}
```

8.5 An example that illustrates how condensed a C statement can be. The bracketing ensures that k has a value from getchar before the while test is made. The way this runs on your machine depends on whether or not a buffer is used.

This function tests the keyboard in a loop of its own, unlike the action of INKEY$ in BASIC, so that nothing happens until a key is pressed. The action also provides for echoing the character to the screen so that you can see what character you have obtained. You can also delete the character with the backspace or delete key.

The next level of brackets is used to enclose the test for a **while** loop. In this case, the test is for the character not being ASCII code 9, the TAB key in most computers. The main action of the loop then follows in the form of **putchar**, a function that prints the characters from the **getchar** buffer. The effect is that this program allows you to press keys and get characters on the screen . Pressing the RETURN key then puts the set of characters on to a new line, and lets you start all over again - unless you pressed the TAB key. If you pressed the TAB key in the middle of a set of characters, only the characters up to the TAB key get printed. If you run the program with the **putchar** statement removed you will see the difference. The point here is that C combines into this very compact statement line the actions that in BASIC would need lines such as:

```
10 K$=INKEY$
20 IF K$="" THEN 10
30 IF K$=CHR$(9) THEN 60
40 PRINT K$;
50 GOTO 10
60 END
```

which is why C is the preferred language for systems programmers. These examples, incidentally, show that the descriptions of standard functions do not always prepare you for what the functions do, because a lot depends on the compiler and the computer. You might expect that the action would be to show each letter twice, side by side, on the screen.

The key to it is the buffer action of **getchar**, because this means that the loop does not run completely until the RETURN key is pressed. Until the RETURN key is used, the **getchar** action puts characters into the buffer, and the loop action with **putchar** empties the buffer after RETURN has been pressed. The presence of the TAB code in the buffer breaks the loop at that point. The use of a buffer for **getchar** follows the example of UNIX, but it's possible that your computer/compiler combination might not use such a buffer, so that the **getchar** action is terminated by the key that is pressed rather than by the RETURN key. In some ways, this is more useful, and if the buffered method is followed it's useful to have another function that will allow unbuffered input.

We can extend this to carry out rather more actions in the loop, as Figure 8.6 shows. The **while** loop starts just as before, but there is no **putchar** action nor semicolon in the **while** line. Instead the start of the loop actions is marked by another curly bracket, and to make sure that you know this is the start of the loop I have indented the bracket. You don't have to do this but it helps enormously in making your programs easy to read, and some compiler editors will automatically carry out this identing when the program is listed.

```
main()
{
char k;
while((k=getchar())!=9)
 {
  if(isalpha(k) && islower(k)) k=toupper(k);
  putchar(k);
 }
}
```

8.6 Adding some function actions in a while loop.

The first action in this loop is a test to find if the character is a letter. This is done by the **isalpha(k)** function - a built-in function on my compiler. This function returns TRUE if the character assigned to **k** is a letter, FALSE otherwise. The other part of the test is done by **islower(k)**, which will return TRUE if the character is a lower-case one. If both parts of this test give TRUE, then, the character is a lower-case letter, and the second part assigns k to the result of the function **toupper(k)** which converts from lower to upper-case. Note how this assignment has to be done – you can't just use **toupper(k)** and then **putchar(k)** because carrying out the function **toupper(k)** does not change what has been assigned to **k**. That's one of the small points that will often trip you up until you become really familiar with C. You can, of course, have statements such as **printf("%c",toupper(k))** that will print the result of the action of the function, but in this example we want to alter the assignment to variable **k** and to do so needs the re-assignment that is illustrated.

As you would expect from the form of the test, this re-assignment is carried out only if the character is a lower-case letter. The result is then printed by using **putchar** as before, and the closing curly bracket marks the end of the loop, with another curly bracket to mark the end of the program. When this one runs, then, the lower-case letters that you type are converted and shown as upper-case when you press RETURN, and you can escape by pressing the TAB key as before.

If you are getting quite confident, then perhaps it's as well to come down to earth by asking yourself what the program of Figure 8.7 might do. The function **isdigit(k)** tests the character assigned to k to see if this is a digit, and will return TRUE if it is. We have declared variables d and m as integers, and made them have starting values of 0 for **d** and 1 for **m**. If the character in k is a number, then, it is converted to number form by the use of **(k-'0')**.

```
main()
{
 char k;
 int d,m;
 d=0;m=1;
 while((k=getchar())!=9)
  {
   if(isdigit(k))
   {
    d=d+(k-'0')*m;
    m=m*10;
   }
  }
 printf("%d",d);
}
```

8.7 A not very serious character to number converter. It might be useful if you like your number written backwards!

This is the shorthand C way of converting from the ASCII code version of k into the number that it represents by subtracting the ASCII code for zero. The effect of this, then, is to convert the ASCII form of a digit into the digit itself. This is multiplied by **m**, and added to the existing value of **d**. Since **d** starts at 0 and **m** at 1, then the result for the first digit will be to assign that digit's number value to variable **d**. The next time round, this is added to ten times the number value, because the line **m=m*10** has made variable m equal to 10, and on the next loop this will be 100, and so on. What this should do is to place the digits into a number variable that can be printed and used as a number. At the end of the loop, the **printf** statement prints out the result of all this. Notice that the action **m=m*10** could be written in the shorter form **m*=10.**

Now brace yourself and try it! Type 123(TAB) and press RETURN, and the screen shows your 123 followed on the next line by 321! This is because the first digit that you type is being treated as the units, the next as the tens and so on; the reverse of the way that we write numbers. If you type something like 12(RETURN) then 34(TAB)(RETURN), then the result is as you would expect from the first time. If, however, you type something like 123(RETURN) 456(TAB)(RETURN) then the result is *not* what you would expect.

By using so many digits you will have overflowed the size of an integer number, and this guarantees that the answer will be incorrect, even if you read it the wrong way round. The moral here is that if you want to carry out conversions like this, you need to use strings rather than the action of **getchar**.

The DO loop

The alternative to the while loop is the do loop whose general syntax is:

```
do
  statements
  while (test);
```

and as you can see the test is made in the same way as for the **while** loop but at the end of the loop. It's slightly unfortunate that the same word, **while**, has been used for both loops, because when you read a C program carelessly you can sometimes be trapped into thinking that the **while** is the start of a loop rather than the end. This can be avoided if you type your loops indented so that the **do and the while** form an indented block. As it happens, the **do..while** loop is not used to nearly such an extent as the **while** loop. Nevertheless it has its uses, most of which are bound up with the fact that the test of the loop is at the end rather than at the beginning, like the REPEAT...UNTIL loop in BBC BASIC. If, for example, you want to create a simple 'Press any key' step, the **do** loop is much more convenient.

The listing of Figure 8.8 shows such a loop, including a function that may not be implemented on your compiler. This is **keyhit()**, a function of the Hisoft **C** compiler that makes a test of the keyboard for one character only, with no buffer action, no looping, and no need to use the RETURN key. Since this is such a frequently needed action, there is probably something of the same kind in your own function library under this or another name.

```
main()
{
 char k;
 printf("\nPress any key...");
 do k=keyhit();while (k==0);
 printf("\nnext step");
}
```

8.8 A 'press any key' step using a do.. while loop. You will need to find what your compiler provides for the keyhit function here.

In this example, the **do** and **while** are in the same line and the loop runs until a key is hit. The **keyhit** action returns 0 until a key is pressed, on which it returns 1. This action, incidentally, does not clear the keyboard buffer and you may find that the character in the buffer will affect some following action if you do not remove it. The point of this example, however, is that the loop in this case has to be tested at the end since **k** has a value assigned to it by the **keyhit** function. This does not mean that the use of a **while** loop is ruled out, because you could use **while (! keyhit())** to test for the absence of a key press. Since the **while** form is neater, it's used to a much greater extent except when the test is one that is on a variable that cannot be assigned with a suitable value until later.

You can, of course, have as many steps as you want between the **do** part of a loop and the **while** test. Figure 8.9 shows an example of this in the form of a very simple integer to binary conversion. By very simple, I mean that this ignores things like sign, and is intended to work with numbers in the range 0 to 32767 only.

```
main()
{
int n;
printf("\nPlease type an integer\n");
scanf("\n%d",&n);
itob(n);
}
itob(j)
int j;
{
printf("\n            ");
do
 {
  putchar(j%2+'0');
  putchar(8);putchar(8);
 j=j/2;
 }
 while (j!=0);
}
```

8.9 An integer to binary converter, using left-shift of the cursor to place the digits correctly.

The principle is that when an integer is divided by 2 it will give a remainder of 1 or 0 which forms one digit of the binary equivalent. The snag is that this process issues the binary digits in the order of least significant to most significant, so that when the digits are printed they will be in the wrong order. It's easy enough to reverse the order of a set of digits if they are in the form of a string, but that's something we haven't looked at yet, and we have to resort to trickery. In this case the trickery is to print a space for the number, print the least significant digit, and then take two spaces backward for the next digit. On my computer, the backspace is obtained by using **putchar(8)** and it's highly likely that your computer uses the same ASCII code for the backspace. The result is quite pleasing!

In the listing, the integer is obtained from the keyboard by using **scanf**, with the **&n** used because **scanf** always requires a pointer. If this were a serious program we would want to test the size of the number at this point, but in the brief example we simply call the conversion function **itob(j)**. This uses **printf** to create a space of suitable size (fifteen spaces), and then starts the **do..while** loop that carries out the conversion. The action of **putchar(j%2+'0')**; will find the remainder after dividing the number by 2, and convert this remainder, which must be 0 or 1, into ASCII code form by adding the code for '0'. This leaves the value of **j** unaffected, and the next line carries out integer division to get a new value for **j** – seasoned programmers would write this as **j/=2.** The loop then continues until the value of j is zero.

After each character has been printed, however, the cursor is moved two spaces left by the two **putchar(8)**; statements, so getting the digits of the binary number into the correct order. In this example, the items in the loop have been put between curly brackets, and each statement is terminated by a semicolon. There is no semicolon following **do**, and the **while** test must be placed outside the curly brackets that enclose the loop actions.

Last word

The **while** loop is one that, along with the **for** type of loop, you are likely to use for most of the types of loop that you need, with the **do** loop being used for just a few special cases. Every now and again, though, you find that there is a special requirement that doesn't seem to fit neatly into the loop arrangements. Usually this takes the form of a test that is included in the loop and that cannot be part of the **while** test. The answer in almost every example is the use of **break** or **continue**, sometimes both. The **break** and **continue** actions can be put into any form of loop, and the **while** or **do** types are no exception. As an example, Figure 8.10 shows a simple character gathering program that rejects any characters that are not letters, and ends when a whitespace character is found. This is an artificial example to some extent because the whitespace character test would be made part of the test in a **while** loop, but as usual it's an example intended to show actions, not to illustrate neatness.

```
main()
{
char k;
do
 {
  k=rawin();
  if (isspace(k)) break;
  if (! isalpha(k)) continue;
  putchar(k);
 }
 while(k);
}
```

8.10 A program that prints a set of characters that do not include any digits, and which ends when a white space characters is used.

The example also uses another function that your function library might not possess, or may have under another name. The **rawin()** function tests the keyboard in a loop and returns with the ASCII code for whatever key has been struck. In some C libraries, this is the action of the **getchar** function, though the buffered form is more common. In the listing, then, the **rawin** function gets the ASCII code, and this is then tested. The first test is for the character being a white-space character (RETURN, SPACE, TAB) and this breaks the loop. The next test is for the character not being a letter, and this causes a continue action. The use of continue at this point means that if a digit or punctuation mark is typed, it will not reach the **putchar** part of the listing, but will cause a return to the start of the loop. Any character that passes both of these tests will get to the **putchar** statement and will cause the character to be printed on the screen. An action like this, incidentally, is not just a useless example, because it's sometimes important to be able to make a word that includes only letters, with no white space or punctuation marks. One application might be for entry of a filename, another might be for the creation of "random" words.

Chapter 9
Arrays, strings and more pointers

Arrays

In BASIC, you have the use of simple variables, such as integers, reals and strings; and you also have one structured variable, the array. By 'structured', I mean that an array name like A means not just a single value, but a set of values that carry distinguishing names like **A(1), A(2)** and so on. C is very well equipped with 'structured' variables, or structured data types, as they are called.

The array is one of these types and in this Chapter we shall look at what it is and what we can achieve with it. As you might expect, an array has to be declared at the start of a program and this declaration will include a name (the identifier), the number of elements in the array, and the type of data that is to be stored. This is just what you would expect from experience of arrays in BASIC. When you use a DIM statement in BASIC, such as DIM N$(20), you are specifying the name **N**, the type (string) and the number of elements (from 0 to 20, a total of 21). The main difference in C, then will be the way in which an array is defined rather than the information that is used.

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

```
int marks[20];
```

This provides the name of the array, which is **marks**, the number of items (20 of them) and the fact that each item will be an integer. This variable declaration can be made along with other declarations of integers in the same line. One important point to note here is the use of the **square** brackets. If you forget that you are writing C and not BASIC, it's easy to refer to an item as marks(12), when you should be using marks[12] instead. The error message that you get when you do something like this will not necessarily remind you of what has gone wrong. Since the array in this example is an array of integers, you can use **scanf** to get each item of the array. One very important difference between the BASIC array and the C array, however, lies in the way that items are numbered.

When you define a BASIC array as A(20), then this allows for 21 items, A(0) to A(20). By contrast, the C array allows for just the 20 that you specified, and these will be [0] to [19] , there will be no [20]. This might not stop you trying to use an item [20], and this is one of the things that you have to be very careful about, because C doesn't usually stop you from doing foolish things, it lets you go ahead and pour garbage into the memory, which you don't find until later! Obviously compilers vary, and some compilers might possibly draw your attention to an obvious error, but in general mistakes like using an array item that hasn't been dimensioned are run-time errors that the compiler can't catch. You must therefore test carefully and build in traps for errors of this type because the memory corruption that they are likely to cause will not point in any obvious way to the source of the trouble.

```
main()
{
int num,marks[20];
for(num=0;num<=19;num++)
{
printf("\nMark %d - ",num+1);
```

```
 scanf("%2d%c",&marks[num]);
 }
printf("\f\n%42s\n","MARKS");
for(num=0;num<=19;num++)
 {
 printf("\nItem %2d got %2d marks",num+1,marks[num]);
 }
}
```

9.1 Filling and printing a number array. Note what happens if a three-digit number is entered.

Figure 9.1 shows an example of this array in use, showing how the array can be filled and how its values can be printed out with suitable formatting. In this program, a set of twenty numbers is obtained. These are assumed to be numbers in the range 0 to 99, but there's nothing to stop you from entering numbers like 5000 unless you incorporate a loop that checks the numbers and calls for re-entry if an unsuitable value is entered. The items are entered in a **for** loop, using principles that should be familiar by now. The prompt line uses **num+1** rather than **num** so that you can have numbers from 1 to 20 instead of 0 to 19. In the **scanf** line, the **%2d** allocates the numbers in two-digit sets. This is not ideal, because it means that if you type a four-figure number, it will be allocated to two sets of marks! For the moment, though, it will serve to keep the numbers below 99. The other point about the use of **scanf** here is that the array *pointer* is used directly, as &marks[num] .

When all of the items have been entered, the screen clears, and the title **MARKS** is printed centred. This is done in the **printf** line by using the control string "\f\n%42s\n". The \f part clears the screen on most computers, and the \n part takes a line down. The next part, the %42s, is the field size number for the word MARKS. Since the field size number represents the *total* size of string that is printed, a word that is of less than this size will be padded with blanks at the left-hand side, in other words, it is right justified. By using a positive number, we force the word to be printed with any of these padding blanks on the left-hand side. The choice of the number 42 with MARKS (which has five letters) means that 37 spaces will be printed to the left of the name, leaving 38 spaces to the right on an 80-character screen.

The method of calculating the correct field size for centring a phrase is to count the characters in the phrase, add the number of characters per line for the screen that you are using, and divide this total by two. If, incidentally, a negative 'field' number is used, the excess spaces are printed to the **right** of the name. This can be useful for spacing the next name, but is not so useful for a heading.

The items of the array are then printed out in order, using the variable name **num** as the array number and **num+1** as the item number. The printing line has made the spacing such that the lines will be uniform for both single-digit and two-digit numbers. This has, as usual, been done by using the specifier %2d in the **printf** line, so that the field allows for two-digit numbers and single digit numbers will be printed on the right-hand side of the field.

The method of using **scanf** with the number specified by %2d makes sure that no number of more than two digits can be entered. This is not always a desirable method of checking, however. The main problem is that if your finger slips and you enter 999 instead of 99, then 99 gets entered in one mark, and the last '9' becomes the first digit of the *next* mark. The program as it stands gives you no chance to do anything other than grin and bear it. This is a compiled program, remember, so you can't just use the old BASIC trick of commanding a GOTO to get you back into the right part of the program.

This method of specification, then, is suitable only when there is no loop involved, so that it doesn't matter if something gets left over. If you want to be able to correct an item *without* running the program all over again, then an IF test (as in BASIC) is a preferable method, or a loop as shown in Figure 9.2. In this listing, which is a modified part of the marks program, the assignment of **scanf is to a variable tmp** that will need to be declared in the earlier part of the program. The point here is that the value of **tmp** is tested in the **do..while** loop, and accepted only if the value is less than 100 (hard luck on swots). This is not quite as staightforward as it might seem, because the **scanf** line has needed a change to %4d to cope with the possibility of a four-figure number being entered.

```
main()
{
int num,tmp,marks[20];
for(num=0;num<=19;num++)
 {
do
 {
   printf("\nMark %d - ",num+1);
   scanf("%4d%c",&tmp);
 }
 while (tmp>=100);
 marks[num]=tmp;
 }
 printf("\f\n%42s\n","MARKS");
 for(num=0;num<=19;num++)
  {
  printf("\nItem %2d got %2d marks",num+1,marks[num]);
  }
 }
```

9.2 A method of testing for correct entry - the do loop will continue until a number of 100 or less is entered, and only then is the array value assigned. Note that this is not a complete program, only a replacement for the reading action in Figure 9.1.

The golden rule is that your program should never bomb out when the unlucky user (you, perhaps) has just entered a lot of data, but this is never easy to achieve for every possible error. If, in this example, you enter a five figure number as a mark, then it gets split among more than one array member as before. This is a **scanf** problem rather than one of program logic, and the answer is to enter numbers in string form and use a function to check and convert – but that's for when you have rather more experience.

Arrays and pointers

In the example of an array in use, we have stuck to methods that were not particularly different from the ways in which an array is used in BASIC. Don't let this fool you, because the way that an array is organised and the ways in which it can be used in C can be startlingly different. The most obvious difference arises because of the way that C uses pointers. Take a look, for example, at the short listing of Figure 9.3 which fills a small array and then prints out the items in two different ways.

```
main()
{
int j,nr[5];
for (j=0;j<=4;j++)
 nr[j]=5-j;
for (j=0;j<=4;j++)
 {
  printf("\nItem %d is %d",j,nr[j]);
  printf("\nItem %d is %d",j,*(nr+j));
 }
}
```

9.3 Illustrating how the array values can be printed by using the array pointer for each value.

We needn't dwell on how the array is filled, or on how it is printed out by using **nr[j]**, but the second **printf** line is certainly different . In this line, the quantity that is printed is *(nr+j), a content of a pointer address **nr+j**, and this gives exactly the same results as the quantity **nr[j].** This is due to a piece of cunning (or economy) on the part of the designer of C , who has made the name of an array identical to a *pointer to the first member of the array.* When you declare **nr[5]** you are in effect creating address space for five integers, and the name **nr** is assigned with the address of the first item. To prove this, add the line **printf("/n%u",nr)**; just before the end of the program, and you'll see a number printed out when the program runs.

This is the pointer address for the start of the array, and if your computer can switch from the **C-compiler** to a monitor you may be able to access this address and look at what is stored in it. Whether you can do this or not, though, the point that is important is that the name of an array is itself a pointer, and we can get to any element of the array by adding the element number to the pointer before using the * sign, as the example illustrates. This is something that we'll make a lot more use of shortly when we come to consider the role of strings in C.

Multidimensional arrays

As in BASIC, arrays in C can have more than one dimension. As always, this has to be declared, and the method of declaration is rather different from the method that is used by most varieties of BASIC. Instead of using a declaration like DIM A(5,4), C uses the form **int a[5][4]**, using a separate pair of square brackets for each dimension number. The items of the array are written in the same way, and the guiding principle is that an array of this type is an array of **an array**. That apart, the use of a multidimensional array is not all that different from the use of such an array in BASIC, as Figure 9.4 illustrates.

```
main()
{
static int marklist[5][3]={
{56,47,62},
{81,65,56},
{74,54,67},
{38,52,46},
{63,50,48}
};
int j,k;
printf("\f\n%s%21s%20s%18s\n","No.","PHYSICS",
"CHEMISTRY", "MATHS");
for (j=0;j<=4;j++)
```

```
        {
        printf("%d",j+1);
         for (k=0;k<=2;k++)
          {
           printf("%20d",marklist[j][k]);
          }
         printf("\n");
         }
      }
```

9.4 A two-dimensional number array in action. In this example, the array is filled at the time of declaration, and the numbers are then printed into columns.

In this example, the array is dimensioned and filled in one operation. This requires the use of a static array, and the form of the initialising is worth noting. Following the declaration is an equality sign and an opening curly bracket. The numbers in the array are then listed separated by commas and between curly brackets. Following each ending curly bracket is a comma until the last line has been filled. The end of the initialising action is then marked by a final curly bracket with a following semicolon. It takes a lot more space to describe than to use. This initialising procedure is not, of course, something that can be used only with two-dimensional arrays, and any variable can be declared and initialised in one operation like this but the use of the curly brackets is confined to array filling. The initialising of a variable in C takes the place of the READ...DATA lines that are so beloved of BASIC programmers. In this example, of course, the initialising has been done for the sake of the example only, because in a real-life program you would enter the data either from the keyboard or from a disk.

Following the declaration and initialising of the array, two more integer variables are declared to act as counters for the array printout. The rest of the routine is concerned with printing the array as a set of marks in different subjects for a group of five student numbers. As is often the case, the arrangement of the printing is the really hard part of this program. The basis for the printing arrangement is the line that prints the marks in fields 20 characters wide. These fields start displaced by one character because of the line that prints the student number by itself at the start of each of the output j loops. The aim in the long **printf** line is to get the subject titles positioned so that they will be reasonably centred on the columns of marks.

The best way of doing this is a very old-fashioned method – the use of layout-paper. This is paper marked with 80 columns across the page (40 columns if your computer uses this screen layout) so that you can position letters and numbers and see how they will turn out. If you take such a sheet and write in a set of numbers using the correct columns, then you can write in the titles above them, correctly positioned.

You can then count up field spaces, remembering that this means the space from the previous last-printed character to the last character of the current word, and note these numbers for your **%s** specifiers. This has been done for the title line with the results shown for an 80-column screen. The two loops then print lines and columns of the numbers, with the inner k loop responsible for the set of numbers in one line. Note that the numbering of j and k is, as always, from zero. The outer loop then moves from one line to the next, using a **printf("\n");** statement to take a new line.

Finally, on this point, if you have strong nerves you can work directly with the array pointers. The substitution in the printing line is:

```
*(*(marklist +j)+k)
```

because the two-dimensional array is an array of an array. To get the first number of the set, for example, you need to use **marklist, and the line above gets the pointer stored in (marklist+j), adds k to this, then uses this pointer to find the number.

Strings at last

A string in C can be regarded as an array of ASCII characters, ending with a zero. This is true in all versions of C, and even in some types of BASIC, but in most forms of BASIC there is a readymade string variable type, marked with the dollar sign, as one of the main variable types. C does not have these string variables readymade and we have to define them for ourselves. A string is an array of characters, and we always define it in that way, as for example **string[20]**. When you write characters for a string in a program, you enclose them with quotes just as you would in BASIC, and you don't have to type in the zero that is always used to end the string. As an example, which is also a very important guide to the use of pointers, take a look at Figure 9.5.

```
main()
{
static char string[]="string test";
int n;
char *p;
p=string;
for (n=0;n<=11;n++)
printf("\n%c, code %d is in %u",*(p+n),*(p+n),p+n);
printf("\n%s",string);
}
```

9.5 A string declared and initialised. The loop prints out the characters of the string to show that ASCII codes are used with a zero terminator.

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

The real meat of this example, however lies in the use of a pointer, defined as **p.** Now in the 6th line of the program, we make the assignment **p=string**, which looks very peculiar until you remember that the name of an array is also the name of the pointer variable for the start of the array. The character array, like any other array, is stored in consecutive addresses in the memory, as this example shows, and the use of pointers is particularly handy just because of that. You don't, of course, have to make use of pointers if you just want to print a string or select one item from it. You can print a string by using **printf**, as the last line of the program shows, knowing that the pointer for the string is represented by its name. You can pick a letter from the string by using the fact that it is an array, so that **string[4]** is the fifth letter (the count starts at 0 , remember). Later, we'll see that it is possible to assign and use a string using only the pointer, with no string variable name at all.

```
main()
{
static char string[]="set of letters";
printf("\n%s",string);
printf("\n Sixth letter is %c",string[5]);
}
```

9.6 Printing a complete string, and selecting a character from a string.

Figure 9.6 shows string selection more clearly. Once again the string is initialised, and the complete string is printed using a **printf** line. Only the array name, **string** needs to be used here, and no square brackets are needed. In the following line, the fifth character in the string is printed by using **string[4]** - remember once again that counting starts with zero. Using the idea that each letter in an array can be located by using its subscript number, you can always assign one array to another variable name. For example if you have string variables **chars** and **string**, both of which are character array types, you can use a routine such as is shown in Figure 9.7 to make a copy of the array **chars** into the array name **string**. This is pretty much the same method as you would use in BASIC to copy one array into another.

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

9.7 Copying one string to another. This has been done in a main program but would normally be carried out by a function. The receiving string must be declared of at least the correct length.

As it happens, you probably have a more efficient method available, in the form of the **strcpy** function in the library. Figure 9.8 shows how this can be included into a routine, and in this example the function **strcpy** has been included. If you need just one library function, it's much easier just to type the source code of the function that you want into your listing than to have to wait for the library to be read each time you compile. The library version makes use of pointers, as you might expect, and it's a good example of just how compact C code can be made. The function returns a character array pointer and because of that has to be placed *before* main so that my compiler can work with it. This isn't necessary if you have another compiler, if you are reading the function from the library as you compile, or if you alter the way that the function is written. For the moment, though, we'll stick with the way it is.

```
char *strcpy(new,old)
char *new,*old;
{
static char *temp;
temp=new;
while (*temp++=*old++);
return new;
}
main()
```

```
{
static char chars[]="Sample string";
char string[20];
strcpy(string,chars);
printf("\n String is %s",string);
}
```

9.8 The strcpy function listed, and its use in a program.

What the function does is to assign a pointer variable **temp** to the same value as the existing pointer for the new string that has been passed to it. The loop then assigns characters from the old string to the temporary one until the zero at the end is found, so making the quantity within the brackets zero. The routine then returns the pointer for the new string, now filled with characters because of the string **temp** that used the same pointer.

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

9.9 A string entry and print routine using the character functions rather than printf and scanf.

Let's get back to the strings, however. Figure 9.9 shows a simple string variable reading and writing program. The word **name** is defined as an array of 20 items of type char. This means that **name** should not contain more than 20 characters (including the terminating zero), but there is nothing to stop you from entering more than 20. This is a feature of C, that you have to build in your own safeguards, the language provides only the minimum necessary. If you enter only two characters from the keyboard, no harm is done, but the rest of the array will be filled with garbage. If you enter 22 characters, the program may crash at a later stage. After the declaration and initialisation steps, the program starts a loop with the line:

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

- which is the sort of thing that gives C a bad reputation with academics. When you unravel it, it's not quite so bad as it looks, and you'll soon learn to shrink lines down to this state. The way to unravel anything like this is to start at the innermost brackets. Within these, you'll find:

```
name[n++]=getchar()
```

and this assigns the character from the keyboard to **name[n]**, and then increments n. Note that when **getchar()** is used in this way with my compiler, no buffer is involved and no RETURN key needs to be pressed. This complete part of the statement is enclosed in brackets, following which is **!='\n'**, testing to find if what is within the brackets is *not* equal to the RETURN key code. The whole of this is enclosed in brackets which follow the **while**. This has the effect of testing if the whole expression is TRUE or FALSE. If the key is not RETURN, the expression is TRUE, and the **while** loop is carried out.

In other words, the character is put into the string and the place number is incremented. When the **\n** code is found, the expression that follows **while** is FALSE, and that's the end of the loop. The whole of this **while** loop is in one line, marked by the semicolon at the end of the line. The two actions that follow are putting a zero at the end of the name, and then making n equal to zero again. Using name[n]=0 actually causes the string to include a **\n** character, but the effect of using name[n-1] doesn't make much difference until the printout stage, when it changes the number of spaces under the printed version. We could, in fact, save a line here by using:

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

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

In C, string actions always look rather complicated because assignment is not quite so easy as you might think. Since a string is a form of array, and there's no method by which you can copy one array into another except character by character, it all looks like hard work. It's really only a problem, however, if you are 'thinking in BASIC'. You can assign a string constant, for example, by using #define in a lot of places where in BASIC you would use a string **variable.** You can also use the scheme that we looked at earlier, of defining a **static char string[]** into which you can assign anything you want at that time. You can also make a pointer point to any string you want, which is probably the easiest way of re-assigning a string name. We don't need to go into this, because it's one function that exists in every C library.

When you really need to use a string variable is when you are inputting or outputting strings, and once again, there is a library routine, **gets(string)** for this purpose. We'll look at these routines later. The thing that you really have to be careful about is any attempt to assign a string to an array which is not large enough, because only a string of at least the same length is compatible. If your strings are arrays of 20 characters, then only another array of up to 20 characters *total* (including zero or any \n character) can be assigned. This is very difficult to get used to when you have been accustomed to the free and easy ways that BASIC has with strings, and it's something that you simply have to come to terms with.

Arrays of strings

In BASIC, you are accustomed to being able to use string arrays, with assignments such as M$(5)="MAY". In C, a string array is an array **of an** array of characters, another two-dimensional array with one of the dimensions being a set of ASCII codes. An easier method is to define a string as an array of characters, and then define another name as an array of strings. Once you have defined your 'string array' name correctly, you can use the string array rather as you do in BASIC. You **must** remember, however, that the rules are rather more strict. Each name in the array, for example, will consist of not more than the declared number of characters, and it's likely that anything following the zero that marks the end of a string will be garbage.

Look for example at Figure 9.10. This consists of a program that will fill an array with names (of up to 20 characters including the final zero), clear the screen, and then print the lot out. We start by defining **name** as an array of ten strings, each of which is an array of characters, up to 20 characters long. Two integers, n and j, are then defined to be used as counters. In the first loop, using n=0 to n<=9 because the array elements are 0 to 9, not 1 to 10, the array **name** is filled with names that you type from the keyboard. Each string is referred to by its two numbers, the place in the array of strings, and the character in each string.

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

9.10 Entering and printing an array of strings.

For example, name[2][4] means character 4 in string 2, remembering that character 4 is the *fifth* character, and string 2 is the *third* string. The zero is inserted as character **name[n][j-1]** after the end of the loop. This is because the counter **j** will have been incremented again at the end of the loop, and using **j-1** gets the zero into the correct position. The screen is then cleared, and the array of strings is then printed on the screen by using the other loop. Each name in the array is obtained, once again, by using its array number within square brackets; name[7] in C is equivalent to NAME$(7) in BASIC. Note that this time, you don't have to use two sets of square brackets.

By specifying that you want to print a string, you have automatically made it unnecessary to specify the second set of square brackets. Just as you could use **printf("%s",title)** to print a string called **title**, which was defined as **title[25]**, you can use printf to print a string which is called, for example, **name[4]**.

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

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

so that any day can be found by use of the array number. These instructions in BASIC are very wasteful of memory, because everything that you have in a DATA line in BASIC is stored in two places while the program is running. The action of READ....DATA in BASIC is really just the initialisation of an array in C, something that we have already looked at, and the program of Figure 9.11 illustrates this action for string initialisation in a form of a routine that you can put into function form and use for your own programs. This time, the array is an **array of pointers**, one pointer for each string, with **no restriction on string length,** because the length of each string is just the length of the quantity to which it is assigned. The name of the pointer is week[], it points to a character type, and its initialisation is carried out as shown. The storage class must be static, if we are to carry out declaration and initialisation in one line, and the new feature is how a set of words, between quotes, can be put into an array.

```
main()
{
static char *week[]={
"Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday","Sunday"};
int n;
for(n=0;n<=6;n++)
 printf("\n Day %d is %s",n+1,week[n]);
}
```

9.11 A program that makes use of the array of pointers to strings that have been declared and initialised in one step.

The method is not so very different from that of Figure 9.4 except that not so many curly brackets are needed for this array of pointers as for the two-dimensional array of numbers. The contents of the array are shown between curly brackets, separated by commas. In an initialisation, there is no need to show the number between the **square brackets** of the name, so this goes in as ***week[].** When the array of strings is printed out, we don't print *week[0], *week[1] and so on, but week[0], week[1] etc. This is because the pointer name is the name of the array item. This is the kind of thing that's always likely to catch you out when you first start writing programs in C, and it's the first thing to suspect if you find that a printout gives you a screen full of gibberish.

Note, incidentally, that each string will end correctly with a zero. You haven't put this in, but it's taken for granted when you use letters between quotes, like "Monday". Remember the difference between single quotes and double quotes. If you specify 'M', then a single character is stored. If you specify "M" then two characters are stored, the **M** and a zero because this is a string.

So far, we have been looking at comparatively short programs. When your programs get to the length at which they take up more than one screen 'page', a printer becomes a more pressing necessity. It's particularly useful if you are using pointers and you are not sure whether you should be using a *x or just x at any particular time. If you can see the declarations at the same time as you look at the lines that are giving you the problems, it all becomes much easier. Another problem is that by the nature of C , you tend to have a lot of nested { and } marks. If you can see these only on the screen, it's very difficult to be sure that each } corresponds to the correct {. Of course, if you planned the program correctly in the first place, you will have checked the nesting on paper.

The problem arises, however, when you have been doing some editing, renumbering the lines, correcting mistakes and so on. At that stage, checking for an incorrectly placed } on the screen alone can be rather a frustrating task. One thing that can make a C program much easier to read is indenting each new { or loop. In this way, sections that are 'compound statements', running as if they consisted of one single instruction, are set away from the left-hand side in a block. This makes it easier to see where the { and } of each block is located. Another advantage mentioned already is that it makes it easier to see if a **while** is at the start of a loop or the end.

More about pointers

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

Before we start on extended pointer exploration, then, recall for a few moments what a pointer is. A pointer is a number that locates a piece of data. A pointer can be defined as a pointer to an **int, char** or any other data type, simple or complex. If the data type is a simple one, the pointer is the number which gives the location of the first byte of that data. If the data type is complex, like an array or a structure, then the pointer gives the address of the start of the array or structure. If we want to make use of a pointer, we must declare its name, and also assign it. We can carry out actions on pointers that include incrementing and decrementing, addition and subtraction of integers, comparison of pointers, and the subtraction of one pointer from another (only for pointers of the same type).

The valuable feature of pointer arithmetic is that C makes automatic allowance for the size of data. If you have an array of characters, for example, then you can define and assign **pc** as a pointer to the first character. Incrementing this pointer, either by using **++** or by adding 1, will get a pointer to the next character. Since each character takes up just one byte of memory, this isn't exactly surprising. If you have an array of integers, however, in which each integer uses two bytes of memory, then changing the pointer by using ++ or by adding 1 will still get the next integer, **even though the pointer has to change by two bytes this time.** This is extremely valuable, because it means that you don't continually have to be worrying about the numbers that you add to pointers. You can, of course, add numbers greater than 1 if you want to get hold of other parts of an array.

The use of pointers in this way, along with passing pointers to functions, is illustrated in Figure 9.12. In this example, two arrays have been declared and initialised. One is an array of characters, the other is an array of integers. The assignments pc=name and pn=data will make the pointers point to the start of each array. Note that this type of assignment is legal because the name of an array is also the value of its pointer. The important difference between the pointer that we assign and the variable name is that the name is a **fixed pointer**. In other words, we can assign **pc=name,** because pc is a pointer variable, but we *cannot* assign **name=pc** because **name** is a fixed amount, the pointer for the start of an array, which cannot be altered except by assigning another array. This important difference is not always well emphasised in books. Once the pointers have been assigned, we can use them in function calls. Two function calls are shown, one to **write(p)** which will print a string of characters pointed to by **p**, and **dates(p,n)** which will print **n** dates, one on each line, pointed to by **p**.

```
main()
{
static char name[]="Ian Sinclair";
/* use your own name here*/
static int data[]={1956,1966,1983};
char *pc;
int *pn;
pc=name;
pn=data;
write(pc);
dates(pn,3);
}
write(p)
char *p;
{
 printf("\f\n");
 while(*p!='\0')
 putchar(*p++);
}
dates(p,n)
int *p,n;
{
 int j;
 for (j=1;j<=n;j++)
 printf("\n%d",*p++);
}
```

9.12 Passing pointers to functions, showing the effect of incrementing pointers.

The real meat of pointer use is now contained in the function definitions, and we'll look at **write** first. The header contains the parameter **p** which will have to be declared before the first curly bracket of the function, because this is the variable that is passed to the function. Since **p** points to a character, it is declared as such. Remember that these names are completely local to the function so that we can change them without altering the quantities that are stored as **pc, pn**, **n** in the main program. Following the first curly bracket, the screen is cleared, and the loop uses putchar() to print a character on the screen.

The character that **putchar** uses is *p, the character that pointer **p** points to. At the start of the loop, **p** takes the same value as **pc**, because this was the value passed to it. In the **putchar()** statement, however, we use ***p++** so that the value of p is incremented by one character position *after* the character has been printed, since the ++ follows the p rather than being placed before it. This will ensure that the next character is fetched when the loop goes around again, and the **while** condition will ensure that the printing loop stops when the terminating zero of the string is found. The **dates** function behaves in a rather different way with an integer also passed to it, and **printf** used to ensure that the date is printed in the form of an integer number. Two variables have been declared in the header, and another, **j** is declared inside the function to be used as a loop counter.

Once again, using ***p++** ensures the correct next number, though this time the memory is being incremented by two units instead of just one.

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

As an example, take a look at the program in Figure 9.13. This contains a function **exchange** which will swap two strings of different lengths, by swapping their pointers. Now it's important to realise that only the pointers are swapped, and the strings remain assigned to their original names, as the program demonstrates by printing out the string names after swapping. If we print out using the pointers, however, the swap will be obvious. The moral, then, is to work with pointers at all times if you are going to make such changes. The routine declares and assigns two strings, **name** and **city**, and the pointers are declared and then assigned. The **printf** lines then show what the pointer addresses are. If it still unnerves you to see these as negative numbers, use "%u" in place of "%d" in the **printf** lines for printing the pointers.

```
main()
{
static char name[]="Ian Sinclair";
static char city[]="Edinburgh";
char *pc,*pd;
pc=name;
pd=city;
printf("\n%d , %d",pc,pd);
printf("\n%s %s","Name is",pc);
printf("\n%s %s","City is",pd);
exchange(&pc,&pd);
printf("\n%d , %d",pc,pd);
printf("\n%s %s","Name is",pc);
printf("\n%s %s","City is",pd);
printf("\n%s %s","String name is",name);
printf("\n%s %s","String city is",city);
}
exchange(x,y)
int *x,*y;
{
 int tmp;
 tmp=*x;
 *x=*y;
 *y=tmp;
}
```

9.13 A program that exchanges pointers by swapping the pointers to the pointers.

The first printing shows the pointer numbers and the names in the correct order. Following the **exchange** function, however, the pointer numbers are swapped, so exchanging what they point to. This appears only if we **printf** the pointers **pc, pd** as strings, not the names **name** and **city** which do not change. Note that the **printf** line uses **pc,pd** and not ***pc,*pd**. Once again, this is because the name of an array *is* the pointer to its starting address. In this context, quantities such as ***pc,*pd** are meaningless unless you want to see the first characters of each string.

The pointer exchange is carried out using **pointers to the pointers.** The pointers to the strings are simply two numbers that are stored in the memory. To exchange them in a function, we need to find where these pointer numbers are, and we can do this by finding their own pointers. This is done by calling function **exchange** with parameters **&pc, &pd**, which are the pointers to **pc, pd** respectively. These pointers are passed to the function as numbers **x** and **y,** and defined as pointers to integers, which they are. The integers that they are pointing to are the pointers **pc,pd.** We then use these pointers to exchange the values of **pc** and **pd.** We cannot simply exchange **pc** and **pd** in a function, because the function works with **local** values only. At the end of the function, any quantities that are passed to the function are restored to normal.

If we work with pointers, however, we can make permanent alterations in anything that these pointers point to. In this case, the quantities that we are working with are temporary values **x** and **y**. These are manipulated so as to exchange **pc** and **pd**, pointers which have not been directly passed to the function. Using pointers in this indirect way is the only method by which a function can make changes in a number of parameters. The routine carries out the swap, and when the main program takes over again, you can see that the pointers have been swapped. The important feature here is that a pointer (for a 8-bit micro) is a two-byte number. You can swap pointers like this around as much as you like, and the action is quick and easy. It's certainly not so easy, and definitely not so fast to try to swap the actual contents of strings around. You wouldn't be advised to try it on strings of unequal defined lengths, either, but when you work with pointers all things are possible. If, incidentally, you want to find where the pointers to the pointers are stored, add a line:

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

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

Arrays of pointers

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

To form and make use of such an array of pointers we have to know what syntax has to be used to refer to pointer arrays. Figure 9.14 shows a pointer array being used to swap pointers round so that the string arrangement is different, using a simple swap routine for three items rather than the complications of a full-scale sort for so few items. Note once again that, contrary to what you might expect, you have to use the address sign, & in front of the pointer array values in order to pass the pointers to the pointer values correctly to the function. Another point to note is that the use of a function will successfully result in obtaining pointers to the pointers, but this is not so simple if you want to carry it out in the main part of the program. The reason is that quantities such as ptr[0] are defined as string pointers, and you can't obtain pointers to them in a straightforward way.

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

9.14 Altering the order of strings in a pointer array.

The only simple assignment you are allowed to make is of another pointer to a string. You can get around this restriction by using a cast statement, something that is very poorly illustrated in most books on C. The use of a cast is to make a quantity become of a specified type, and the syntax *for my compiler* is **cast(type)quantity**. This is not the normal syntax, and you will normally find that the word **cast** does not appear, so making the action seem rather mysterious. Figure 9.15 illustrates this action in operation. In this example, the strings have been printed in quite a different way that illustrates the usefulness of pointers to pointers. The line:

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

will have the effect of making the quantity **ptr[0]** temporarily into an integer, and then taking the address of this integer and assigning it to integer pointer s. For your compiler, the more likely syntax would be **s=&((int) ptr[0])**. Using the correct **cast** expression allows **s** to carry the address of **ptr[0]**, and the printing loop can now make use of ***s++** both to print the value and to increment to the next string. This is simple because the string pointers are held at consecutive addresses.

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

9.15 Using the cast construction to convert a pointer into an integer.

Another aspect of the use of pointers with string lists is how easily an item can be located. This makes for very efficient and short routines for such actions as finding the day of the week from a number. An illustration of this use is shown in Figure 9.16, which starts with the definition of a function. The function is of type **char** and it will return a pointer, because this is what the name of one item will be, one of the array of pointers. The header will put in an integer which will consist of a number in the range 0 to 9. A number such as 10 will count as 1, because only the first character will be accepted by the main program call to **getchar**. This number is declared and treated as an integer in the function, and the pointer array ***day** is declared and (since it is static) assigned. The assignment of the zero position is made to a message. After this, the last line of the function returns the selected string. The variable c is used as a selector in the line:

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

so that if c is less than 1 or more than 7, the string day[0] will be returned, giving the message 'no such day'. For numbers between 1 and 7 inclusive, the correct day of the week is returned counting Monday as day 1. The 'no such day' message is returned for numbers 0,8,9, but if you type 10, then you will get Monday because only the 1 of 10 has been accepted by **getchar**.

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

9.16 A day number to name converter using pointers.

There is an old programmers proverb that says a fool can make more mistakes than a wise programmer can anticipate. There is also a users proverb that says a programmer can make more mistakes than anyone can anticipate.

Chapter 10
Menus, choices and files

Most varieties of BASIC allow a neat and simple method of programming menus, using the ON K% GOSUB type of command. Since a menu is a very common feature of a lot of programs, it's time that we took a look at how such a system can be programmed in C. The key to simple menu programming is the **switch** statement, which is C's equivalent of ON K% GOSUB. Suppose that you have a list of items on your menu, with each item numbered in the usual way. You then use a keyboard read function to input a number. Suppose, for example, that you use the **getchar** function, which is built-in to many compilers so that it does not need to be read from the library. You can then program:

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

and so on, with the closing curly bracket at the end of the list. The function **getchar()** will obtain values that are ASCII codes for numbers such as 0,1,2, and so on, so that getchar()-48 converts to number form as we saw earlier. This allows

switch to select the command which appears after the same number in the list that follows **case**. You could equally easily omit the conversion and use lines like **case 49**:, but this is undesirable because most compilers will allocate memory for the missing cases 0 to 48, so wasting space.

```
main()
{
int j;
printf("\f\n%42s\n","MENU");
printf("\n%10s","1. Start new file.");
printf("\n%10s","2. Add items to file.");
printf("\n%10s","3. Delete item.");
printf("\n%10s","4. Change item.");
printf("\n%10s\n","5. End program.");
printf("\n%10s  ","Please select by number- ");
do
 {
 switch (j=getchar()-48)
  {
   case 1:printf("\n start new file");
   break;
   case 2:printf("\n add to file");
   break;
   case 3:printf("\n delete item");
   break;
   case 4:printf("\n amend item");
   break;
   case 5:printf("\n end of program");
   break;
   case -38:break;
   default:printf("\n No such item- %d please try again.\n",j);
  }
 }
 while (j>5||j<1);
}
```

10.1 An example of menu construction using switch and case. Note the importance of the break statements.

In a real program, each of these options would consist of a function name, or a statement, and there would be some form of error-trapping to ensure that the correct range of number choices was not violated. For an illustration, we can substitute simple **printf** statements, as in Figure 10.1. The important point is to understand why the **break** statement has been added in each line.

In the example, the screen is cleared by the **\f** part of the first **printf** statement. This prints a title, and the fielding command %42s has been used for an 80-character screen. Note that when anything like this is done, the message must be separated by a comma from the fielding. If you use **printf("\f%42s\n MENU");** you will probably see a set of gibberish characters appear preceding the word MENU, though compilers differ in this respect and you may find that you *can* use such a syntax. The menu items are then printed, with a number allocated to each item. Notice the use of "%10s", the C equivalent of TAB(10). You are asked to choose by number, and a large **do** loop starts.

The first loop action is to obtain the number choice, using **getchar** in the usual way. This, however, appears as the second part of a **switch** statement, showing once again how C can make these very useful compound statements for an action that would require several statements in BASIC. What follows lists the numbers and actions. Each number is followed by a colon, then the action or actions that must be carried out. In a 'real-life' program, of course, each case would summon up a different function rather than the simple **printf** that has been used here. Each choice has simply caused a phrase to be printed in this example, because the aim is just to show what the **switch** statement does and how it is programmed. To see why we need the **break** statements, try omitting one or two.

You'll see that this has the effect of allowing more than one answer to be printed. The **switch** statement allows you to select one item, but when the action returns, it will move to the *next case statement* , no matter what number has been used. Unless you actually want the next **switch** line to be carried out, then, you must make this next statement a **break** to allow the rest of the **switch** sections to be skipped. Notice, too, that we can cater for a selection which is not in the range that **switch** allows. This is done by the **default** item, and it's a very handy way of ensuring that the entry range is checked and something sensible done for each possible answer.

Since all of this is enclosed in a **do..while** loop, the selection is repeated until a choice in the correct range is made. The message is printed by the function, and the **do** loop along with the **default** statement ensures that the choice can be made again until the number lies in the correct range. It's not quite so straightforward as it seems, however. If you simply add a **do..while** loop to an existing menu, you need a quantity to test at the end, and this can be done only by using a variable to store the value obtained from **getchar**. In the example of Figure 10.1, the integer j has been used. At the end of the loop, the value of **j** is tested. This contains the statement (J>5||j<1), with the vertical bar signs used to mean logical **OR**, so that the statement in brackets tests the truth of 'j greater then 5 OR j less than 1'.

The trouble is that a simple loop can result in the default message being issued twice, once for the incorrect number, and once again for the RETURN key, depending on how your computer handles its keyboard buffer. The RETURN key returns ASCII 10, and 10-48 gives -38, so this is assigned to j after the first default message, causing another message - but there are no more characters left now in the buffer. Now this could be sorted out by a bit of machine code that clears out (or flushes) the buffer, but there is a simpler 'all-C' solution, which is to make a **case -38:break**; to detect this event and ignore it. This is one of the delightful things about C, that there is so often a way out of difficulties that doesn't involve digging into the machine code. This is important, because the whole point of using C is to be able to write programs that will run on any machine that can read the disks, and if you are writing C for CP/M use, then this could mean a very large variety of machines indeed. Because of this, it's best always to avoid machine code unless you are certain that your programs will be used only on machines for which the machine code is compatible. The set of **switch** actions must end with a closing curly bracket, and the whole program ends as usual with the final curly bracket.

A suitable case for treatment

The control for **switch** does not necessarily have to be defined as an integer because, as we have seen previously, a character is entered as an ASCII code which is an single-byte integer anyhow. You can, therefore, use a character to control a **switch**, as is illustrated in Figure 10.2. In this example, the variable **s** is of type **char**, meaning a letter, and the **switch** statements are set up for letter testing. We can still use **s=getchar()** to get the character from the keyboard, however. This is because, once again, the language does not make any rigid division between characters and integers– the main difference as far as the computer is concerned is that a character is stored in only one byte of memory, and an integer requires two bytes. Note that a character is referred to by using its key, within *single* quotes, such as 'S', 'A' and so on. This is something you constantly have to remember, because using double quotes, such as "S", "A", means a string which consists of the letter code *and a zero.*

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

10.2 Controlling the <u>switch</u> action with a character variable.

Once again, in this program, the use of a **do..while** loop can cause problems on machines that use a keyboard buffer, in particular the repetition of the 'type a letter' message. This time, the method that has been used to prevent this is different. It is certainly possible to use an extra case line to detect the character '\n', but this does not prevent the prompt from being printed twice. The test at the start of the loop, however, along with the assignment to '@' before the start of the loop does what is needed. Assigning the value '@' to s before the loop starts prevents the extra **s=getchar()** step from being used. If the loop returns because the @ key has not been pressed, the value of s cannot be @, so the extra **getchar** will read the ENTER code of '\n', and allow the program to operate normally. These steps may not be needed with your computer or (possibly) your compiler.

This type of character input can be improved by using some of the built-in library functions, and one of these can also be used to get over the RETURN key difficulties. The improved program is shown in Figure 10.3. This time, the test for escaping from the loop (the @ character) is made at the start, using a **while** loop. You have to be careful how this is done, with the **s=getchar()** step enclosed in brackets and made not equal to '@', and the whole expression in brackets for the **while** statement.

If you get these brackets wrong, such as by using **while(s=getchar()!='@')** then you will find that s will be assigned with either 0 or 1, depending on whether the key was '@' or not respectively. The inner brackets in the listing are essential to ensure that s is assigned with the character, rather than with the 0 or 1 which is the result of the test using !=. By putting this test into the outer brackets, you make it apply to the while loop rather than to the assignment. In the loop, two tests are then made. The first test uses the isspace function, which is TRUE if s happens to be a space, the newline character or a TAB. In this example, it's the newline we are trapping, and the effect will be to continue if the character is a newline.

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

10.3 An improved character-testing program, using some character testing and altering functions.

'Continue' used *in any type of loop* means that the rest of the loop will be skipped, and the loop is restarted. If the newline is found, then, the loop returns for another **getchar**. The next test uses function **isalpha**. By using this in the form if **(! isalpha(s)),** we get a TRUE answer if the character is **not** alphabetical. For this event, we print out the 'not a letter' message, and continue to get another letter. If character s has survived so far, we then use function **tolower(s)** so that any upper-case letter is converted to lower-case. This avoids the problem of entering an upper-case letter like A, E, I, O ,U and being told that each is a consonant. On my compiler, all of these functions are built in, and don't need the library to be searched, but you may need to use a **#include** form of statement to read in these functions.

One last point about **switch** is worth mentioning if you have been used to writing BASIC programs that make selections from strings entered at the keyboard. The expression that follows **switch**, within brackets, must give a single integer. You can't for example, make **switch** work with strings, except to recognise the first character of a string. If you have to work with strings, then a program like the one in Figure 10.4 will be more suitable.

```
main()
{
char j;int n;
char command[6];
printf("\n Please type command");
printf("\n CLS,UP,DOWN,LEFT,RIGHT");
do
 {
 n=0;
 while ((j=getchar())!='\n')
  {
   command[n]=tolower(j);
   n++;
  }
 command[n]='\0';
 if (!strcmp(command,"cls"))putchar('\14');
 if (!strcmp(command,"up"))putchar('\13');
 if (!strcmp(command,"down"))putchar('\12');putchar('\10');
 if (!strcmp(command,"left"))putchar('\10');
 if (!strcmp(command,"right"))putchar('\11');
 printf("\351");
  }
  while(command);
 }
int strcmp(s,t)
/* graphics block shape*/
char *s,*t;
{
 while(*s==*t)
 {
  if (!*s) return 0;
  ++s;++t;
 }
 return *s-*t;
}
```

10.4 Menu action with strings. The switch action does not apply, and tedious comparison lines are needed.

The name **command** is defined as a string of up to six letters, and it is filled with characters by using a loop which contains **getchar**. This uses testing for the ASCII code of 13 to check the use of the RETURN key. The use of **getchar** in a program like this is not ideal, because it echoes to the screen and uses buffered input. My compiler supports another more useful input function, **rawin** which does not place anything on the screen.

For some purposes, particularly graphics programs in which pressing an answer should not show on the screen, this kind of action can be useful. There is virtually nothing fixed in C for input or output, however, mainly because in the early days of C all such actions were governed by the UNIX operating system. Your compiler may support the types of input/output that are used in UNIX, or it may use quite different actions that are better geared to the needs of a small computer. Whatever type of input is used, the command word that is obtained is then compared with a list of keywords by using **if** tests with **strcmp**. There is *no way in which you can make a direct comparison of one string with another in C* , so that lines such as:

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

are *never* valid. This is something that takes a long time to get used to when you have programmed previously in BASIC or even in PASCAL. The **strcmp** function, which is certain to be in your function library, does the comparison character by character, and returns a number whenever two characters are unequal. If the strings match perfectly, then the function returns 0. We have to test for NOT **strcmp**, therefore, using the ! sign.

In this example, the standard **strcmp** function has been put into the listing so that you will not need to extract it from your own function library. The program example will then do things like clear the screen and move a cursor about by using long commands such as UP, DOWN, LEFT, RIGHT. This assumes that the codes for these actions are the ones that have been shown here, and you will have to make whatever adjustments are needed for your own computer.

The important point to note is that the numbers used in the **putchar** statements are not in denary, but in octal (scale of eight). The number denary 8 thus apears as 10 (one eight and no units) and denary 10 as octal 12 (one eight and two units). If you can make use of a lower-level input function to replace **getchar** then you should be able to arrange it so that the words do not appear. You might also want to make another comparison test to get out of the loop, because as the program stands the loop is endless and will require the use of a BREAK key or similar to stop it running.

Data files

Now that we have used arrays and tackled the construction of menus, it's likely that you'll want to record data information on to disk. At this point, it becomes more difficult to give specific examples, because how you create and use disk files varies a lot from machine to machine unless you are using a standardised operating system such as C P/M, MS-DOS, FLEX or UNIX. The easiest way is to start at the beginning, and look at what is involved in the standard C methods of recording and replaying a list of integers which will be held in the computer as an array. Figure 10.5 shows what is involved. The integers are generated in a loop, which simply gives all the multiples of two up to 100. Once this array has been generated, the recording file is opened by using the line:

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

in which fp is a **pointer** to the start of the file, "intfil" is a filename that will be used on the disk, and **"w"** means write.

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

10.5 Disk filing for an array of integers. Your machine may require further preparation to use disk files.

The whole action could have been carried out in one loop, but I wanted to separate the generation of the numbers from the filing routine so that it would be easier to adapt the program for something more useful. The file must be closed by using **fclose(fp)** after writing. If there has been any other file called **intfil** on the disk, it will be deleted or renamed or the new file left unrecorded, depending on how your machine operating system handles such events. As usual, C does not dictate how these things are done.

To read the integers back in a more controlled way, we should write a reading program in C – and that's the next step. One possible reading program is illustrated in Figure 10.6. This one prepares in the usual way, and opens the file using

```
fp=fopen("intfil","r")
```

with the "r" (not 'r') meaning "read" in this case. The loop is performed as before, but this time **fscanf** is used, and the syntax is not the same as that for **fprintf**. The reason is that an array is being filled, and **fscanf** needs a pointer to the position in the array. Now the name of the array, **b** is the pointer to its first item, b[0], so that if we use **b** by itself in the **fscanf** instruction, all numbers will be read into the first item. To make the pointer shift to the correct item, we use **b+n,** so that the correct number of address bytes above the pointer start **b** is being used. Remember that when you add to a pointer in this way, the number that is actually added is a calculated number, taking into account the type of data. For example, an integer uses two bytes.

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

10.6 Reading the file of integers that was prepared by the previous program.

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

The use of **fprintf** and **fscanf** is just one of a set of ways of using disk filing. Figure 10.7 demonstrates two other standard disk-filing functions of C that can be used, **putc()** and **getc()**. As the names tell you, these are character functions, but this description can be very misleading, particularly as applied to **getc()**. The action of **getc()** is to return an integer, which can, of course, be regarded as a character in ASCII code. The important point is that you can assign **getc()** as an **integer** or directly as a character, but it's better always to assign it as an integer. The reason is that you generally use **getc()** in a loop that will continue until the end-of-file character is found.

```
#define EOF -1
/* or whatever your computer needs*/
main()
{
char a[51];
int n,j,*fp;
n=0;
fp=fopen("newchar","w");
while ((n<=50)&& ((j=getchar())!='0'))
 {
 putc(j,fp);
 n++;
}
fclose(fp);
getchar();
printf("\n press RETURN key to read file");
getchar();
fp=fopen("newchar","r");
if (fp==0)
 printf("\n no such file");
 else
 while ((j=getc(fp))!=EOF)
  {
printf("%c",j);
  a[n++]=j;
  }
 a[n]='\0';
 fclose(fp);
 printf("\n%s",a);
}
```

10.7 Using putc and getc in single character files.

Note that this is **"w"**, a string, not **'w'**, a character. The function **fopen** is the name of a standard UNIX function and will be in your library – it was built-in to my compiler so that no **#include** was needed for my listing. Once the file has been opened, the array can be recorded by using another loop, with a variation on **printf** being used in the writing process. The function **fprintf** is used very much like **printf**, but with the file-pointer as the first of its arguments. Once again, **fprintf** was a built-in function on my compiler but might have to be read from the library for your machine.

The EOF character in most varieties of C is -1, which in integer form consists of two bytes, hex FFFF. If you read **getc()** as a character, it will deliver only one byte, and the end of file character cannot be read. That's usually one fruitful cause of program crashes. Another is to gather the characters into a string and forget that there must be a zero at the end when the string is printed!

Looking at the program of Figure 10.7, then, the assignments are made as usual with character string a, and the others integers. As before, the file pointer has been defined as a pointer to an integer. The counter n is initialised to zero so as to make a count of the maximum number of characters that can be entered.

The file is opened, and a **while** loop starts. In this loop, the number of characters that can be entered is limited to 51 (from 0 to 50), and the loop condition also includes a **getchar** step that will allow entry of a character and detect a zero being pressed. This allows you to type letters as you please, using spaces, newlines or whatever – until the zero key is pressed and followed by RETURN. The file which was opened at the start of the program is then used by **putc(j,fp)** to place the character corresponding to integer **j** in the file which is pointed to by **fp**.

This loop continues until a '0'(RETURN) is entered or until the maximum permitted number of characters has been entered. The file is then closed, and the program hangs up, waiting for you to press the RETURN key. For my computer, and perhaps also for yours, a **getchar** step has been put at the end of the file-writing part to trap the RETURN character that has been used to enter the zero and is still stored in the keyboard buffer.

When you press RETURN, the program then continues, opening the file for reading. Now it can happen that you don't have the correct disk in the drive when you are reading a file, and the next part of the program shows how to deal with this contingency. The pointer **fp** will be zero if no file exists, so that testing for **(fp==0)** allows you to print a message. In a real program, of course, you would want to return to the waiting step if the disk turned out to be the incorrect one, but in this example, the program simply stops if the **newchr** file is not on the disk. If the file is found, then a **while** loop reads it until the EOF character is found. The EOF has been defined at the start of the program as -1, the correct EOF for my computer.

We could, of course, have used -1 in place of EOF, but if you use **EOF** and **#define**, it's much easier to change a program so as to run on another machine (or another variety of C). The **getc()** function is assigned to the integer j so that the EOF can be detected, and the conversion to characters is done simply by using **a[n++]=j**, in which the character is placed in the array of characters and the place number incremented. When the loop ends because of the EOF character, the '\0' is added to make the array into a true string. The file is closed, and the string of characters is printed. Now you can start condensing the size of the program by merging actions in the usual C way.

String files

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

The program is illustrated in Figure 10.8. It's considerably longer and looks more complicated than any of the C programs that we have looked at so far, and there are several new points embedded within it. The first point is that two routines from the standard C library have been included in the program.

```
#define EOF -1
#define NULL 0
extern char *fgets();
main()
{
int n,*fp;
char str[80],*sp;
 n=1;
 fp=fopen("strfil","w");
  do
  {
   getline(str,80);
   fprintf(fp,"%s",str);
  }
 while (++n<=10);
fclose(fp);
printf("\n Press RETURN to continue\n");
getchar();
fp=fopen("strfil","r");
 if (fp==0)
 printf("\n No such file \n");
while (sp=fgets(str,80,fp))
 {
  if (sp==0) break;
  printf("%s",str);
 }
 fclose(fp);
}
getline(s,n)
char s[];
int n;
{
 int c,i;
 i=0;
 while (--n>0 && (c=getchar())!=EOF && c!='\n')
  s[i++]=c;
 if (c=='\n') s[i++]=c;
s[i]='\0';
return(i);
}
char *fgets(s,n,fp)
char *s;
int n,*fp;
{
 static int c;
 static char *cs;
 cs=s;
 while (--n>0 && (c=getc(fp))!=EOF)
 if ((*cs++=c)=='\n') break;
*cs='\0';
return((c==EOF && cs==s)?NULL:s);
}
```

10.8 Creating and printing a file of strings.

These library routines have been taken almost directly from the illustrations of library routines in the sourcebook of all C wisdom, the book by Kernighan & Richie called *The C Programming Language.* You will quite certainly find such routines in your own library, but the listings have been included here for the sake of showing just what these routines do. The **fgets** routine has been defined as **extern** because my compiler requires that any function that does not return an integer must either be placed before **main()** or declared as **extern** before **main** and then placed following **main()**. The **getline()** function will get a string named s from the keyboard, and return the filled string, along with an integer equal to the string length, which need not be used.

The important point about the routine is that the string length is limited, and this will ensure that the array declared size is not exceeded. When you use strings, it's important to make sure that a string is not overfilled, because this can create the most remarkable garbage when you try to use such strings.

The first part of the program opens a file called 'strfil' which is intended to take a number of strings, counted by the **do..while** loop from 1 to 10, ten strings in all. Each string is obtained from the keyboard by using **getline(str,80)**, the library function which assigns the string to the name that is supplied, limits the string length to 80 characters (in this case) and returns an integer equal to string length. The string is then saved to a disk file by using the usual **fprintf** routine. Once all of the strings have been read from the keyboard and filed, the file is closed, and the first part of the program ends.

The replay starts with the 'Press RETURN key' step that we encountered earlier. The file pointer is allocated for a read file, and tested in case the file does not exist. Once again, no attempt is made to return to the waiting loop at this stage. The file reading loop makes use of the function **fgets**. This takes three parameters, **str,n,fp,** meaning the string, number of characters and file-pointer respectively. The function will read the file whose pointer is **fp** and return a string of up to n-1 characters from the file. Each string is then printed, and the printing line uses **%s** to specify a string, but no **/n** to force newline. This is because the strings already have newline characters included when they are put into the file

So far, so good, and for the moment, we'll ignore the library functions except to point out that they have to start with a declaration of type unless this is integer, and the asterisk which shows that **fgets** returns a pointer.If you hate to have a mystery left unexplained, I'll deal with the **return** line of the **fgets** library routine. The **test?actionl:action2** line is a way, as we have seen, of choosing to return one quantity or another. If the test is TRUE, then actionl is taken, if the test is FALSE, then action2 is followed. In the example of *fgets(s,n,fp), the line is:

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

so as to select which quantity to return.

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

More structured types

We have come quite a long way in looking at examples and applications of C, but there are still plenty of topics to get to grips with. One of these is records, something that is not easy at the best of times, and more difficult if you have only ever programmed in BASIC. A **record** is a collection of items of data, which may all be of the same type or, more usually, of different types. What makes these items into parts of a record is that they are related.

To take an example, suppose that you wanted to keep a record of membership of the local Football Club. You would need the name and the address for each member. These would be strings, arrays of type **char**. You might also want to keep year of birth (because Juniors pay a reduced fee, and very senior members pay only entry fees) and year of joining (members with ten or twenty years membership have special privilege years). All of these last three items could also be stored as strings, because string entry and storage is easier. There might also be an entry for fees due (a real number in a real-life program) and whether paid or not to date.

Now all of this data constitutes a **record** because for each person, the subject of the record, all the items belong together. It would not make much sense to keep a file of names, one of addresses, one of year of birth, and so on, and yet this is the way that we are often forced to keep such records in BASIC.

The alternative in BASIC is often to pack all the data into one string of set length, and to make up a string array. C allows you to define what will go into a record, and then to create an **array of** records. Obviously, the ultimate aim of such an array would be to record it on disk, something that is relatively straightforward because C allows you to specify a record as a variable type. The type of variable that is used in C for a record is called a **structure**.

We'll start by considering what we need to do in Figure 10.9 to declare a **structure**, using as an example the Football Club illustration above. The program would start with whatever was needed in the way of **#define** lines , functions, and external variables, but the important part is what follows in the **struct** declaration. The name of the record is given as **footballclub.** This is a reminder only, because though we could use this as a variable name, it's rather unwieldy. The name that is used here is sometimes called the 'tag' of the structure. The structure **footballclub** is declared, and what follows within curly brackets must be a list of the **fields** of the record, meaning the items that make up the record.

```
struct footballclub {
  char name[20];
  char address[40];
  char birth[5];
  char join [5];
  float fees;
  char paid;
  }F;
main()
/*as usual from now on:/
```

10.9 Declaring a structure type. This has to be carried out before the start of the main program in which the structure will be used.

I have typed these indented, with one item per line, to make them more obvious. Like any other declaration, the items could be grouped with all the **char** names following the **char** heading, separated by commas. The name and address fields are both strings, but with different numbers of characters. The two years are also taken as strings, with the dimensioning for five characters in the year because there will be four digits and the '0' which marks the end of the string. If you don't dimension adequately, the program will compile and run, but the results will be decidedly odd! The fee amount should be a 'float' number- in a program which was seriously intended to keep records of this type, the amount of the subscription would be calculated from a formula, and printed when required, but in this example, I have made it an entered float item.

If your compiler does not deal with floats then you will have to replace these references with ints. This is a good programming exercise, and if your compiler handles long ints, then the precision of the arithmetic will be better. The letter **paid** is of type **char**, and will be used for a 'Y' or 'N' reply, because the subscription will either be paid or not- this club doesn't allow instalment payments! The end of the definition of the fields of this record is marked with the usual } sign. All of this definition occurs before the start of the main program, and following the curly bracket that ends the structure definition, we must have a semicolon. If there is only a semicolon, then a structure can be declared later by using a line like:

```
    struct footballclub F;
```

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

The main program then starts, and the example stops here, because what follows depends on what you want to do, and there is no point in having large examples. The important point to note is how the inputs are assigned, as Figure 10.10 shows. For the name entry, for example, we can use **gets(F.name)**. This calls function **gets()**, and assigns the string that it gets from the keyboard to variable **F.name**. This is the way that we can select one item (or field) of a record, using the structure variable name, then a full-stop, then the item name. This syntax lets you assign to an item in a structure or print an item. Later on, we'll see that if you want to do anything more complicated you need, as always, to use pointers. The rest of the information is then entered in the same way, and when you want to print the details on the screen, you use the same syntax of structure name-dot-fieldname. For example, you would use a line such as:

```
    printf("%s\n",F.name);
```

to print the name that had been entered.

Filing structures

The structure in C is so useful as a way of packing information into groups that we obviously need some way of recording structures on disk. It would be pleasant if we had a structure filing statement that allowed a complete structure to be put on to disk simply by using the structure name, as we can in PASCAL. This, however, can't be done in C, and we have to record the items of a structure one by one. Though this could be done as part of a main program, it's much more likely that we would want to make the structure filing routine part of a function.

```
/*getting items into fields*/
printf("\n Name please\n");
gets(F.name);
printf("\n Address..\n");
gets(F.address);
printf("\n Year of birth?\n");
gets(F.birth);
/*an so on*/
```

10.10 How the items of the structure are referred to at the input stages.

The important point here is that you can't pass the name of a structure to a function and expect it to do anything about it. You can, however, pass a **pointer** to a structure by using the **&** sign with the structure name. This is needed so often that C has a special way of indicating the items in a structure by way of the pointer. For example, if **sp** is the pointer to a structure, then **sp->item** will refer to the field item in the structure. The **->** sign uses the minus and greater-than signs together.

```
/*called by recfil(fp,&F) */
recfil(fp,sp)
struct footballclub *sp;
int *fp;
{
 fprint(fp,"%s\n%s\n",sp->name,sp->address);
 fprint(fp,"%s\n%s\n",sp->birth,sp->join);
 fprint(fp,"%f\n%c\n",sp->fees,sp->paid);
}
```

10.11 Using a pointer to a structure, and how items within the structure are referred to by pointer->name.

A sample portion of a structure-filing program is illustrated in Figure 10.11. We assume that a disk file is opened for writing with file pointer **fp**. The setup of the structure is the same as before, as is the entry of information. After the Y/N information on payment of fees has been entered, a function **recfil(fp,&F)** is called to place the data of the structure on file. This function uses the file pointer for the disk, **fp**, and also the pointer address **&F** for the structure. The meat of the program therefore lies in the function **recfil**, which uses two parameters. One parameter is the filepointer, fp, an integer, and the other is the structure pointer sp which is declared as **struct footballclub *sp**.

This declaration is that **sp** is a pointer to a structure of type **footballclub**. The fields of the structure are then sent to the file, using **fprintf** statements. The name and address strings are sent first, using "%s%s" as the specifier for the two strings, and with **sp->name , sp->address** as the separate fields. The other fields of the structure are dealt with in the same way, remembering that **sp->fees** is a float, and **sp->paid** is a single character. Since the items that are to be recorded are stored in a buffer until the buffer fills or the file is closed, you don't necessarily hear much activity from the disk at the time when this function runs.

Reading back structures

Reading back a file of structures from the disk normally uses **fscanf**, but you must remember that this function works with pointers. In addition, **fscanf** will take the end of a string as being the first 'white space' in the string, meaning the first blank or any other character which does not 'belong' in a string, such as the TAB key or the space key. This makes **fscanf** more suited for files of integers, or of strings which can be guaranteed to have no spaces in them, but it's not very useful for the type of string that we might want to read, with names and addresses.

Fortunately, the library contains the useful **fgets()** function, which is very similar to **gets()**, but with subtle differences. This function can be used to read back all of the recorded strings, and will not give trouble if any 'white space' is found in a string.

That doesn't mean that everything is plain sailing, because when you use a library function you have to read the small-print (or its listing) to see just what it will do with the data.

The similarity between **fgets** and **gets** as defined in Kernighan & Richie is close, but one difference is *very* important. Whereas **gets** will read a string of characters, including the RETURN/ENTER character, and then replace the RETURN/ENTER character by a zero to act as string terminator, **fgets** does not do this. The **fgets** function reads a string until the '\n' character is found, and then adds a zero to the end of this. This makes the string longer. For example, if year of birth is entered as a string of four characters, it will be recorded as five characters (the '\n' being the fifth), and will be returned into the program as a string of six characters in all, including the '\n' and the '\0'. This means that we have to be careful about dimensioning the strings that we shall read into, because it's easy to fall into the trap of assuming that the string we read back will be exactly the same as we recorded.

This behaviour of **fgets** can be changed by decrementing **cs** in the **fgets** function before it is equated to zero, and you might want to do this and put the result into your library under a slightly different name.

The other point to watch is that **fgets** takes three parameters, the string name, a string length number, and the filepointer. The string-length number decides how many of the characters of the string are read, and the function reads characters until this number is exceeded or until a newline character is found. Unless you are *very* sure of your string lengths, it's better to provide generous values of length, so that the newline character always ends the reading action.

If, of course, all of the strings were tested for length before recording, there's no objection to counting them back precisely. What you need to remember, however, is that the number that you provide for string length in **fgets** must be the complete string length, including the ending zero and newline character.

With these warnings in mind, we can now look at what is needed to read back the kind of string file that was created by a program that used a section such as was illustrated in Figure 10.11. It's likely that we might want to read the structures back into an array, and this will mean that an array of structures must be declared. This is easier than you might expect – all that is needed is to use something like **F[2000];** in place of the **F;** at the end of the structure declaration. The important point is how each structure field in each item of the array should be referred to. A section of a suitable reading program is shown in Figure 10.12.

```
/* all placed within a FOR loop*/
fgets(F[j].name,20,fp);
fgets(F[j].address,40,fp);
fgets(F[j].birth,6,fp);
fgets(F[j].join,6,fp);
/*and so on*/
/*can then use*/
printf("%s\n",F[j].name);
/*and so on*/
```

10.12 Entering items into an array of structures.

The structure has been named as **F[2000]** earlier, and a separate count number could be included in a separate file, as a useful way of ensuring that the writing and the reading programs do not get out of step. In BASIC, this number could be read and then used to dimension the array. In C this type of thing is not so easy, because the structure has to be dimensioned **before** the main program starts. If the declaration is made before the start of the main program (that is, all of **struct footballclub** but without the **F[2000]**) then the declaration of the name (F) and the dimensioning could be done in a function, with the structure not used in the main program. This function would have to be called after the count number was loaded from disk.

When the array has been read in, the program section illustrated here prompts for the name that is being searched for. This is entered, using a function called **getstr(s)**. The reason for not using the normal **gets(s)** is that **gets(s)** will always replace the '\n' character by a'\0', and to match the string that we are entering, we need it to contain both of these characters. The function **getstr** is illustrated here; it's not a normal part of your library. In addition, strings have to be compared, and this cannot be done in the familiar BASIC IF A$=B$.. way. The name of a string in C is simply a pointer to its first character, and you can't expect the pointers to two different strings to be equal. Any comparison such as:

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

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

If a structure name by itself, such as **F[0]** is used in a function, it has to be represented by its pointer, but when parts of a structure are being read, as in **F[j].name,** these can be used directly. The **fgets** lines read in all of the parts of each structure, after which the file is closed. The data can then be displayed using the same **F[j].name** syntax.

Working with structure files

The part-example of Figure 10.12 showed the construction of a reading part of a file that would create an array of structures. A normal action from then on would be to pick out one record, or to sort the records into alphabetical order. Now picking out one record is fairly simple, as the program section of Figure 10.13 shows. The records are assumed to have been read as an array of structures, using **fgets()**, which will result in each string ending with a '\n' and a zero.

```
/* count is number of records*/
/*read into array F[] */
printf("\n Please type name required \n");
getstr(s);
x=false;
/*defined as 0, true as 1*/
for (j=0;j<=count-1;j++)
 {
 if (strcmp(s,F[j].name)) continue;
 else;
  {
  printf("%s\n",F[j].name);
  /* and so on...........*/
  x=true;
  }
 break;
 }
if (x==false)printf("\n Name not found.");
/*continue main*/
/* getstr() function is modelled*/
/* on fgets.*/
char *getstr(s)
char *s;
{
static int c;
static char *cs;
cs=s;
 while ((c=getchar())!=EOF)
  if ((*cs++=c)=='n') break;
 *cs='\0';
 return s;
}
```

10.13 A fragment of program that illustrates how one record can be picked out from an array of structures.

The test for equality of strings is used to make the loop run faster by using a **continue** in the **for** loop if the strings are not equal. When a matching string is found, the **else** section runs, printing the details for the selected name. Before the loop started, integer x was made equal to 'false' (zero), and if a string match is found, this integer x becomes true, and the loop breaks. In this way, the loop runs fast until a matching string is found, and then breaks immediately afterwards. The integer x is used after the loop ends to print a suitable message if no matching name has been found.

Sorting a file

In BASIC, there is nothing that corresponds to the structure. This makes actions such as sorting very tedious, because each field of a record has to be represented by an array item or as part of a string. Sorting is never easy, but in C you do at least have the advantage of a sort routine in the library. The illustration of Figure 10.14 shows a Shell-Metzner type of sort routine used to sort a list of records in alphabetical order of names as typed in the file. This is a simplified type of Shell sort, adapted from a version that I wrote in Pascal- it should compile even on fairly small compilers that don't accept many nested loops.

```
/*read in with fgets into  */
/* array F[j] as usual */
Fp=F;
y=1;
while (y<count)
 y=2*y;
do
 {
  y=(y-1)/2;
  it=count-y;
  for (i=1;i<=it;i++)
  {
   j=i;
   do
   {
    z=j+y;
    if (strcmp (F[z].name,F[j].name)<=0)
```

```
          {
           swap (Fp+z,Fp+j,sizeof(struct footballclub));
          j=j-y;
          }
         else j=0;
         }
        while (j>0);
       }
      }
     while (y!=1);
     /* can now print sorted list */
```

10.14 A Shell-Metzner type of sort routine for an array of structures. In this example, the structures will be sorted in alphabetical order of the first name in the structure.

As before, we assume that the program has read the structures into an array. Because it's more convenient for the sort routine, the array numbers start with 1 rather than with zero, but that's the only change up to the point where we take up the listing. In the structure declaration, the usual F[2000]will have been supplemented by *Fp, making Fp a pointer to a structure. Simply declaring that Fp is a pointer, however, doesn't make it point to anything, and the statement **Fp=F** is needed to make Fp a pointer to the start of structure F. This is a very important step, and omitting it is one of the most common errors in the use of pointers. If your pointer hasn't been set to point at something, then trouble, in the shape of a major program crash, can't be far behind. Why do we need the pointer anyhow? The answer is that in the sort routine, we shall want to change the pointers to different members of the array. Changing pointers involves exchanging only two numbers, rather than the set of strings (or whatever else is used) in a structure. For this reason, it's fast and simple.

I won't go into details of how the Shell-Metzner sort works, because it's a standard routine that you can find described at length in many other books. The important features of the sort (starting at the statement **y=y+1** are the test and exchange steps. The test uses the standard library function **strcmp** as you would expect, with **F[].name** being used as a basis of comparison. The exchange step uses the library function **swap** (not listed in detail here), and the parameters that are supplied make use of the string pointers, along with the **sizeof** statement, as noted in Chapter 5. The swap routine exchanges pointers, if need be, and when the sort is completed, the structures will be in alphabetical order of names.

The important point is that, because of the **Fp=F** step, you can print out this new order using **F[j]**, you don't have to use pointer **Fp** unless you want to. This is the value of altering pointers in this way, because the pointers can be altered in a subroutine and the alteration will affect the result of a printout in the main routine. In this example, the whole sort routine is, unusually, in the main program, simply to avoid the problems of passing parameters until you have seen an example of the straightforward version.

Record nests and choices

So far, each record that we have illustrated has consisted of items that are simple variables. We can, however, use records that consist partly or completely of **other records**! Structures which are a part of another structure are called 'nested' structures, and typically they are used to hold details of an entry. We might, for example, have an entry called **birth** which would require the details of day, month and year of birth. This could be provided by making **birth** a structure in itself, with day, month and year items of that structure. Figure 10.15 shows how this provision for nested structures can be used.

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

```
/* nested structure example */
struct name{
 char sur[20];
 char frn[20];
 };
struct dob{
 int day;
 int month;
 int year;
 };
struct person{
 struct name memname;
 struct dob birth;
 char phone[16];
 }member[max];
main()
/*main starts here*/
/*to refer to an item, we use*/
/*lines such as the following */
printf("\n%s",member[j].memnam.sur);
/* gets surname of member */
printf("\n%s",member[j].memnam,frn);
/* gets forename of same member */
/* and so on */
```

10.15 An example of the formation and use of nested structures.

The important feature now is that a reading or printing line uses the full title for each field and subfield. For any **member** whose surname we want, we have to specify:

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

using the main structure title, the substructure title **(memnam)** and the field title of **sur**. Each entry, replay and print will be done in this way.

The union

A **union** is a 'hold-anything' variable, one that can be used to hold a character, integer, string or whatever we like. It sounds marvellous, but in fact it's not used as much as you might expect. A **union** has to be declared in very much the same way as a structure is declared, using a pattern of the form shown in Figure 10.16. In this example, the type union is declared, with the pattern name of **boss**. The union can contain a character, a pointer, or an integer named j or k.

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

10.16 The declaration of a type union, and a simple example of its use.

Note that this is one *or* another. A structure, by contrast, contains all of the types that are specified, the union contains any one. In the lines that follow, a type is assigned and its value then printed out. The important point is that you can assign only one value at a time, and you must select the correct name, such as **chief.c** , **chief.j** or whatever is needed.

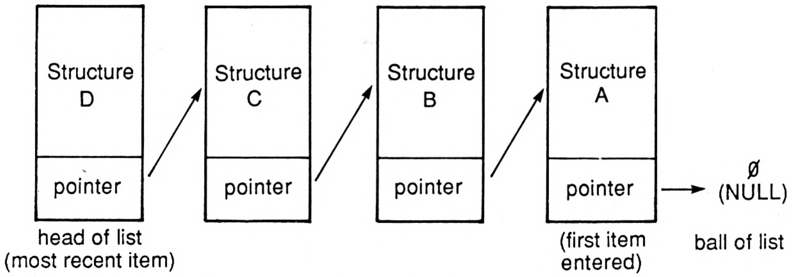
When a union variable is declared, it will reserve as much space as is needed for the largest of its possible contents. If you make a union type, for example, which contains a character, an integer, and a four-character string, then the string is the longest member, and will make the union four bytes long, assuming that the total string length is four. A structure would need one byte for the character, two for the integer, and four for the string, a total of seven bytes.

If you attempt to print out data which has not been assigned to a union, you will get garbage. For example, if you have assigned a string to pointer **s** in the union, then trying to print out a character **chief.c** or an integer **chief.j** will produce results which may be useful, but usually are not. In this particular example, the attempt to print **chief.j** will usually produce the pointer address **chief.s**.

Linked lists

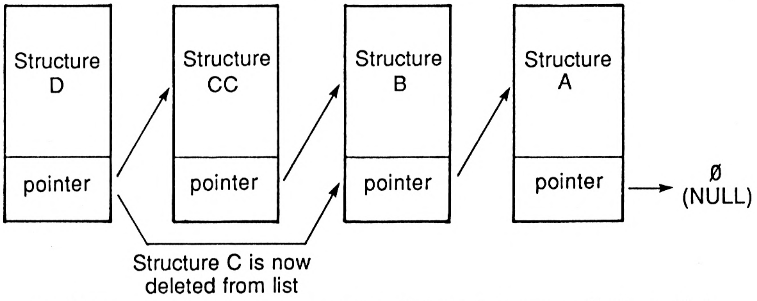
Suppose that you defined a structure which consisted of an integer number and a pointer. Now the most important feature of a pointer is that it can be made to point to something that may be anywhere in the memory of the computer. Because this is possible, we can make the pointer in the structure point to the next structure, even if this means a structure which is not the next one that you enter, or even the previous one.

If you make up a set of structures like this, you don't need an array. Each structure contains a pointer to the next structure so that if you can locate the first structure, you can get to any other, swinging like the legendary Tarzan on ropes of pointers from structure to structure. Figure 10.17 shows in diagram form what this is all about. A sequence like this is called a 'linked list'.



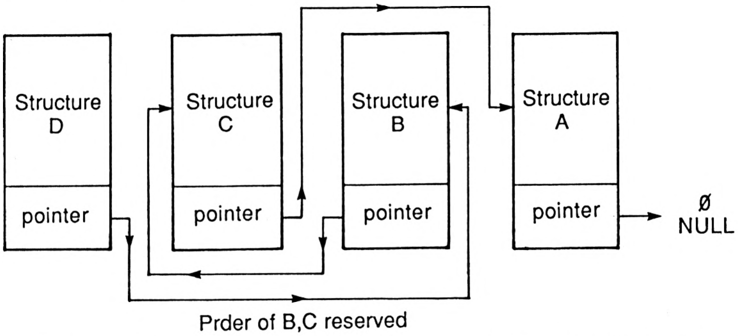
10.17 The general form of a linked list.

What advantages would such a set of structures have? Well for one thing, it becomes very easy to 'delete' a structure. All you need to do is to alter the pointer that points to the item and make it point to the next one instead (Figure 10.18). Having coped with that idea, how would you reverse the order of two structures? Figure 10.19 shows the principle in diagram form, requiring three pointers to be changed. What we have to do now is to see how some of this paper talk can be transferred into a working program.



10.18 How a structure can be deleted from a linked list.

Figure 10.20 shows a typical example which is deceptively simple. Only the main part of the program has been shown here, because when all the necessary functions are added to make it run, the result is rather massive. In this part, then, we define a structure called **record**. This structure contains an integer **daily**, and a pointer **next**. What makes **next** rather different from any pointer we have looked at so far is that it's a pointer to type **record**.



10.19 How to reverse the order of two members of a linked list.

In other words, part of this structure is a pointer to another structure of the same type. Variable **cash** is then defined as a structure of type **record** which contains daily, an integer number, and **next**, a pointer. The variables **first** and **p** are also

```
struct record{
 int daily;
 struct record *next;
 }cash,*first,*p;
typedef struct record *rec_p;
main()
{
int j,x;
p=cast(rec_p)calloc(100,sizeof(struct record));
first=NULL;
/* NULL = 0 */
do
 {
 printf("\n Today's number - ");
 scanf(" %d",&j);
 p->daily=j;
 p->next=first;
 first=p;
 p++;
 }
 while (j!=0);
/* this completes input */
/* output is as follows */
 while (p!=NULL)
 {
 printf("\n%d---%d",j,p->daily);
 p=p->next;
 j++;
 }
```

10.20 Creating and printing out a linked list. Note the order of the list, with the last item at the top.

defined as pointers to a **record**, then the program itself starts with **main()**. This starts by declaring **j,x** as integers, and then finding a value for the quantity p. We need p as a pointer to where each structure is going to be stored, and to get this free space, we use a library function **calloc(number,size)**. The parameters to **calloc** are the number of structures that we would want to use, and the size of each structure.

Rather than count up the size of a structure, we can use **sizeof** to get this number. The snag in this action is that **calloc** returns a character pointer, and we want a pointer to type **record**. This is arranged by two important lines. One is the **typedef struct record *rec_p** which follows the declaration of the structure. The main use of **typedef** is in situations like this along with cast. The statement **typedef** allows us to declare that a word represents a data type. It doesn't allow us to create any new data types, but it *can* define a type that will be accepted by the **cast** statement. In this example, the word **rec_p** is being defined as meaning a pointer to type **record**, the structure. By using **rec_p** in the **cast** statement, therefore, we force the value of the pointer that is returned by **calloc** to be of the same type as a pointer to the structure.

You may think that all of this changing of variable type for a set of numbers that are all integers is rather tedious, but programming can be a lot more tedious in a language which doesn't allow conversions! Remember that the appearance of the word **cast** is rather unusual and the action is carried out on most compilers by typing the line as shown *without* the word **cast**.

The next step is to make **first** point to NULL. There are no values to point to yet, so this pointer points nowhere. A loop is then set up. The preset limit to size for this set of structures is set by the value of size of 100 entries that was used in **calloc** and entry can continue until you enter a figure of zero, or until the memory is full. You get no warning if you exceed the limit that has been set by **calloc**, and if you do exceed it, you may suffer no ill effects, or you may find that the whole program crashes, and your data with it.

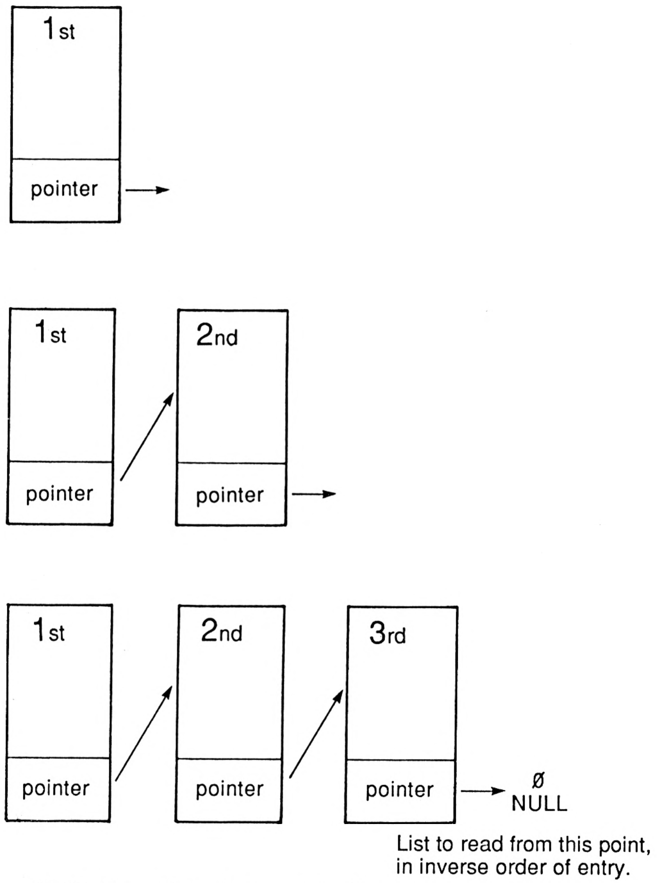
Now what happens in the loop is vital to the way that the list is constructed, and you need to follow it very carefully. I think it's a lot easier if we can think of numbers for the pointers, and so I'll assume that the pointer NULL is zero (true), and that the other pointers will be 40974, 40978, 40982... and so on (possibly true, depending on machine, and it helps you to understand it). Let's take a walk through the first loop round. The value of pointer **p** has already been allocated by **calloc**, and its type is pointer to **record**, so that we can use quantities such as **p->daily** in the same sense as **cash.daily**. We can imagine that **p** carries the number 40974. This will now be used to store the cash amount, **p->daily**, obtained as integer **j** from the **scanf** line.

Notice, incidentally, the blank space preceding the %**d** in scanf. This is put in to ensure that the RETURN character does not cause an endless loop. The pointer quantity **p->next** for this structure is now made equal to first, which is NULL. This is the way of signalling that there is no following structure. Pointer **first** is then made equal to P, assumed to be 40974. The value of **p** is then incremented using **p++.** This makes **p** change from 40974 to 40978, because the structure contains one integer (two bytes) and one pointer (also two bytes), a total of four bytes.

So much for the first loop. What happens in the second? We pick a new P allocation, assumed 40978 this time. Once again, we store a cash quantity, and the **p->next** pointer is this time made equal to 40974, the pointer to the previous entry. Pointer **first** is now made equal to the current value of **p**, which is 40978.

If you look at these steps in diagrammatic form, Figure 10.21, you can see that the list is not growing in the way that you might expect. The 'top' of the list is zero, followed by the item that we entered last of all. Its pointer always points to the next one down, the previous item. The list ends with the one that points to **the first entry**. If any more proof is needed, take a look at what the last part of the program, following the zero entry, prints out. Enter items like 11, 22, 33 and so on that you can recognise.

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



10.21 The list shown in diagrammatic form.

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

What is more important at this point, then, is learning how to operate on the lists so that you can search through a list. The importance of all this is that a lot of computing languages exist in the memory as linked lists. A line of BASIC for example, is usually a structure that consists of a pointer (to the next line), an integer (the line number), and a string of up to 255 characters (the line statements). Languages such as BASIC can therefore be written in terms of a language such as C , or if you are really masochistic, you could try writing a C compiler or interpreter in C - but not yet!

Figure 10.22 shows how you can search through a list for an item. This is a fragment of the program only, and you will need the same additional functions as previously. You need to know, of course, what you are looking for, whether it's the item with a value of 64 or the first one whose value is less than 9 or whatever criterion you adopt. You need not work, of course, with integer number values in the structures, as long as each structure contains some data that you want to use along with a pointer to the next structure. The aim is to find the entry for a given identification number, but you could, of course, easily adapt the program to find whatever feature you wanted of the structure you had defined. The item-finding part has been put into a loop with the **while** condition left to you so that you can write in whatever you need.

```
/* searching a linked list */
/* x is range of numbers */
do
{
p=first;
printf("\n please select item number\n");
scanf(" %d",&j);
if (j>x||j<1) continue;
```

```
      else
      {
       printf("\n%s%d%s","Item ",j," is ");
       j=x+1-j;
       for (y=1;y<=j;y++)
        p=p->next;
      }
      printf("%d",p->daily);
      }
     /* while condition here */
```

10.22 A program fragment that shows how to search a linked list.

In the loop, you are asked to provide an item number, which should then be tested for range. If the item number is in the correct range, the corresponding item is found. This is not entirely straightforward because of the 'upside-down' nature of the list. The **scanf** function returns the item number as integer j, and this is tested for range, using **continue** to ensure that the loop will return to its starting point if the number is not in range. For a number that is in range, the next step is to adjust the variable value, using the expression:

```
    j=x+1-j
```

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

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

Chapter 11
The library

You will quite certainly by this time have gathered that the library that comes with a C compiler, or which you buy separately, is a very important part of your C writing system. The compiler itself often includes some of the most-used routines, the kind we've referred to as 'built-in', but many of the very important routines, particularly those that manipulate strings, are included as part of the library. Your programming in C depends very heavily on how well you understand and can make use of these functions in the library, because you would normally expect to write none, or very few, functions for yourself in most types of programs. In this Chapter then, we shall look at the kind of functions that you might expect to find in a library and how they would be used.

Obviously, though most libraries are based on the foundations laid by Kernighan & Ritchie and will possess many functions that are common, there will be quite a lot of variation from one library to another, particularly in order to exploit the hardware of different machines such as graphics or sound capabilities.

Given, then, that you have a library of functions on disk, what should you do to make the most effective use of it? The first thing to do is to print out the whole library. Even if you have no printer (and it's rather difficult to think that you would have a disk system but no printer), then this is a case for borrowing one, no matter how old or slow. Only when you have a printout of all your library routines can you make really intelligent use of the library routines. In particular, you need to know the type of each function, what manipulations it does, and what it returns.

In this Chapter, we'll look at some types of library routines to illustrate what to look for. This will not be a list of the routines in one library, nor listings of routines themselves, but rather an analysis of what library functions are all about.

Library function structure

Though the standards of C are well established, library functions vary from good C to poor C, and in some cases unintelligible C. You very often find, for example, that different routines have been written by different authors, and have been written in very different ways. This is a contrast with C itself, which is an excellent example of a language written throughout by one hand.

One point that shows up inconsistency is the way that a function header is written. You will find that a function such as **fgets** will be written as **char *fgets** so as to indicate that such a function returns a character. It follows that for consistency an integer function should be headed by something like **int strcmp,** but you may find that several of the functions in the library carry no type indication at all. This means that these functions are of type **int**, because an int type need not be declared. It's always better, though, if the type is indicated in the header. Another point is that many libraries employ header labels of their own (by using a **typedef** statement), even when these labels are really of the standard types. Though a good case can be made for a header label like nil to mean that the function does not return anything *useful*, the function does nevertheless return something, usually an integer, and this should be made clear.

The important features of a function, its type aside, are then the parameters that must be passed to it, as shown in the header, and what it returns, as shown by the **return** statement or by pointer manipulation within the function. If you are examining a function to see what it does then these are the features that you must look for. Another item that turns up with some compilers is the variadic function. A variadic function is one that can take a variable number of arguments, unlike the normal type of function which must use only the arguments that are listed in the header. A function of this type is marked out by the use of the word **auto** in the header, and conventionally by the use of an argument counter variable argc and an argument vector variable **argv**.

The argument counter has a value that is one more than the actual number of arguments in the function, and the argument vector is a pointer to an array that consists of the arguments. These forms of arguments are also used more conventionally for command line arguments, so that a program can be called with its name and one or more arguments. In this respect, you will need to follow closely the method that your own compiler follows in dealing with either type of call.

As a simple example, Figure 11.1 shows a variadic function that also illustrates how the **argc** and **argv** variables are used. The function **addit** will produce the sum of the numbers that are included as an argument list in brackets following the function name. The variadic function is placed ahead of the main program, and its header contains the type, **int** , a name for the parameter list **numbers**, and the word auto that marks this out as a variadic function.

```
int addit(numbers) auto
{
 static int argc,*argv,sum;
 sum=0;
 argc=numbers/2-1;
 argv=&numbers + argc;
 while (argc--) sum+=*argv--;
return sum;
}
main()
{
    printf("\n%d",addit(1,3,5,9,13,15));
    }
```
11.1 An illustration of a variadic function. The same techniques are also used for putting in command-line arguments to the main program.

In the function, the variables **argc**, ***argv** and **sum** are declared as integers, meaning that **argv** is a pointer to an integer in this case, the first of a set of integers. The quantity **sum** is initialised at zero, and then the values of **argc** and **argv** are found. Now to understand what happens here you need to know what is being passed to this function. The header contains the word numbers, but because of the use of **auto** what is passed is a number of bytes, the number of bytes, plus one set more, contained in the arguments.

Since these arguments are in this example integers, then there will be two bytes per argument number. Dividing **numbers** by two gives the total number of integers, and subtracting one gives the correct number of argument numbers. The setting of **argv** is much easier, because **&numbers** gives the start of the array of integers. Remember though that the first member is a 'dummy' one, the extra integer in the count. This convention is derived from the one used in command-line arguments, in which the first argument is the name of the program itself. From this point on, the program is simple – the count is decremented in the **while** condition and the variable **sum** is added to sum plus the value of ***++argv**.

Note the order here- the pointer **argv** is incremented before being used, so that we skip over the dummy value at the start of the loop. Finally, the variable **sum** is used to return the sum of the numbers. Though this example is a very simple one, it contains all the information that you need to make effective use of variadic functions (if your compiler supports this type) and for command line arguments if this is supported. In general, you will probably find that either one or the other will be supported.

Input/output

The input/output functions are very important parts of a library, because they should be closely geared to the computer that the compiler runs on. The specification of C does not tie the compiler writers to any specific input/output routines, and though there are several that are considered classics that must appear in any C library, others are very much at the discretion of the compiler writers. In particular, C provides for no 'machine-level' input or output. By this I mean the kind of single key input like INKEY in BASIC that puts no cursor on the screen, echos no character to the screen, and uses no buffer, or which clears out its buffer after use. Such a raw input is particularly useful for 'Press any key' responses, and essential if you want to make use of a C compiler for creating games.

As an example of such a function, Hisoft-C for the Amstrad machines has **rawin()** which obtains the ASCII code for whatever key is pressed, with no echo to the screen. The corresponding output is **rawout()** which will send a character to the cursor position on the screen. This once again can be very useful, particularly for games or for some graphics work within business programs.

The next functions in order of complication contain the **getchar()** and **putchar()** that we have used extensively in the course of this book. These functions both use the standard input/output, meaning the keyboard and the screen, but other functions at this level are more likely to use files. As it happens, many compilers will support a method of redirecting **putchar** to a file by using the **>** sign and a filename following the program name.

For example, if you call your program by using the name **data**, then using **data>dfile** will cause all the output that would have gone to the screen to go instead to the file called **dfile**. Don't expect the smaller compilers to include this feature, however. For most purposes, inputs and outputs from or to files have to use the specific functions for these purposes. These files will be opened and allocated pointers that are then used in the input/output functions. You will probably find, however, that your compiler uses the pointer number of zero for the standard input and output of keyboard and screen.

A good example is provided by the simple library functions **putc(fp)** and **getc(fp)**, which may be built-in to the compiler rather than in the library. Both are of type integer, and if the number 0 is used in place of a file pointer, the input or output will use the keyboard or screen respectively, unless your compiler adopts a different scheme for standard input and output. Both of these functions operate a character at a time, but the action of **getc** is one that has to be watched. As noted earlier, this returns an integer, so that if you are using **getc** in a loop testing for EOF, then unless **getc()** is assigned to an integer, the EOF can never be detected. Figure 11.2 illustrates these two functions being used with standard input and output.

```
main()
{
char c;
while((c=getc(0))!='\n')
putc(c,0);
}
```

11.2 Using the file pointer zero to mean standard input and output.

On my compiler, **getc** uses buffered input so that whatever you type when this program runs, up to the RETURN key, is echoed when the RETURN key is pressed. The use of filepointers 0 in this example then makes these two functions behave exactly like **getchar()** and **putchar().** One other in this set is **ungetc(c,fp)** which will put character c back onto the file pointed to by **fp**. The result of this will be that this character will be the next one to be read by another use of **getc**. The function can deal with only one character at a time, and is used by scanf so that some care is needed if you need to use **scanf** followed by **ungetc**.

Normally, a call to **getc** following **scanf** is needed before **ungetc** can be used. In the normal use of all three of these input/output functions, you would open a file with **fopen**, using as arguments a filename and the mode. This function is taken straight from UNIX, and the mode is a string which can be "r" for read, "w" for write, or "a" for append. The "a" mode allows the creation of a new file without error messages (recall Chapter 10), and also the addition of data to the end of an existing file. These mode string letters can be followed by a "b" to mean a binary file (containing any number from 0 to denary 255), and/or + to mean read *and* write.

Your compiler will probably include some method of dealing with correct appending to binary files which present special difficulty because of the lack of a conventional EOF character. You will normally find that opening a file with any variety of "w" mode will cause any existing file of the same name to be deleted, but for systems running under CP/M, the file may become a .BAK file in the usual way. Files are closed by using the **fclose(fp)** function.

Writing programs in C would be rather a burden if we had to depend only on these character level inputs and output, so that libraries invariably contain more complex functions. Just how complex depends on how extensive the library is, but here again there will be some hardy annuals that are virtually a part of the language. Main among these are **printf** and **scanf** which are the most important of all the input/output functions, and used in practically every C program. In particular, these are the functions that are normally used for printing or inputting numbers and in small libraries there may be no other functions for number input at all. If your compiler supports floating point numbers, however, it's likely that there will be several functions, possibly at different levels of complexity, to input and output floating-point numbers and integers.

There are versions of **printf** and scanf that use file pointers so that input/output is to or from a named file. In addition there are variations that make use of string input or output. In other words, the printf can be into a named string instead of to the screen or to a file, and a read using **scanf** can be from a named string instead of from the keyboard or a file. Care needs to be taken if the **sprintf** function is used to write into a string that the declared number of characters of the string variable is not exceeded.

The small libraries devote most of their attention in this respect to string input and output. The function **gets(s)** will get a string from the keyboard, using string pointer **s** , and **puts(s)** will send the string whose pointer is s to the screen. These functions are illustrated in action in Figure 11.3, and their action in this listing is exactly the same as that of the character-level functions **getc** and **putc**, with the difference that a string has to be declared and dimensioned.

```
main()
{
 char string[80];
 gets(string);
 puts(string);
}
```

11.3 Use of puts and gets with strings.

Both of these functions are a special case of **fgets** and **fputs**, which perform the same string actions with a named file. The arguments for **fgets** are the string pointer, number of characters and filepointer, and the string is terminated when the RETURN key is used, or when one less than the permitted maximum number of characters has been read. If for example, we use **fgets(string,80,fp)**, then 79 characters could be read, leaving space for the terminating zero as the 80th character. If the string is terminated by the RETURN key, then the '/n' character in the string is automatically replaced by a zero. As we have discussed, you can re-write this function so that the zero is added after the newline character if this suits you better.

Character handling functions

Since single character operations are so important in C it follows that any library will contain a large number of functions that manipulate, test and change characters. We have used some of these functions in the course of this book, and many of them are used to such an extent that they will exist as built-in functions on some compilers. Of these functions, the types that test a character will return an integer that can be tested for TRUE or FALSE, and those that make some change to a character will themselves return a character. We can start by looking at some of the character tests that are found in virtually any C library.

The character test functions are all based on the ASCII code for the character, and are used to determine the type of the character. For example the character can be tested for being an alphabetical letter, or for being either a letter or a number. These tests can be used for checking text, or for checking mixtures of digits and letters as might occur in hex numbers. Other tests can check for control characters, punctuation, graphics characters or whitespace. These tests are useful if you are writing interpreters or compilers, favourite occupations for C programmers. You can test also for the character being a denary digit, of ASCII code, lower-case, upper-case, or printable. These tests can be of considerable use in word processors, spreadsheets, databases and other programs in which the user might enter commands. For some programming work, you might want to use a function that will check if the character is part of the valid hex number range of 0 to 9, A to F.

The character changing functions will consist of three fundamental actions and probably some others that will vary from one library to another. The fundamental functions are the case-changers and the ASCII mask. The case-changers, as the name suggests consist of one function that will change any lower-case letters to upper-case, and another that changes any upper-case letters to lower-case. These functions should have no effect on digits, punctuation marks or control codes. The ASCII mask function will make sure that a character has an ASCII code in the range of 0 to 127 denary by removing any set bit 7 in each byte. This is an action that is common in BASIC interpreters, just to give one example, when the last or the first character of a reserved word is stored with the high bit set - in denary terms meaning that 128 has been added to the ASCII code.

What else your library provides in the way of character test and change depends very much on the size of the library and the purposes for which it is intended. For some purposes, for example, you might want to test for different groups of characters, like all of the arithmetic signs, or to make more specialised changes, like code conversions, particularly for monetary signs other than the dollar. As always, even a small library will probably provide more character functions than you are likely to need and if you can list these functions then you can also use them as models to construct more specialised functions of your own. Where a compiler is designed to run on a specific machine or family of machines there are likely to be some functions that are particularly useful for that machine or machine family, and these should be used with some caution because you will have to replace them if you need to adapt your programs to other machines. At the rate of computer development we have become used to, it would be a foolhardy programmer who believed that he/she would never have to adapt a program to another machine.

String functions

Because the string is not really a data type in C, all of the string handling in a program must be carried out by the use of functions. Any good library must therefore include a large number of string routines, and some libraries make a point of trying to include functions that correspond to the string handling of a good BASIC. Even the smallest library, however, must include the important fundamental functions for string comparison and string copying. These are functions that the former BASIC progammer finds very hard to become used to, because it's difficult for a BASIC programmer to remember that the 'string' name in C is just a pointer, so that actions like string=first and if (str=="end") are quite ridiculous in C, because what you are testing is that one pointer number is the same as another. The standard string comparison function is **strcmp(a,b)** which returns 0 if the strings are identical.

This means that a lot of tests have to be formulated in a negative form such as **if (!strcmp(x,y)...** so as to get a -1 (TRUE) inside the brackets if the strings are identical. The comparison function is also useful for other purposes, however because it will also indicate the alphabetical order of the strings if they are not identical. If the number returned by **strcmp** is positive then the first string of the pair (a in the example) should come after the second string in an alphabetical index. In other words, if **strcmp** returns a positive value, the strings need to be swapped in an index order. If the value of **strcmp** is negative, then the strings are in correct order. A variation on **strcmp** called **strncmp** will check only a given number of characters in each string.

Along with **strncmp** you will find **strcpy**, taking the form **strcpy(string1,string2)**. The action of this function is to make a copy of string2 and call it string1. Like many other string routines, it assumes that a suitable array size has been declared for **string1**, because there is no check in the function that this has been done. This function performs the very important string assignment action, the equivalent of BASIC's A$=B$. There is also a number-limited version of this action in the form of **strncpy**, of the form **strncpy(string1,string2,n)** which will copy n characters only.

On some compilers, this function is not quite so simple as it sounds, and has to be used with some care. If the string that is being copied contains less than the number 'n' of characters, then the copying action is just as it would be with the simpler **strcpy** function, but with the remaining spaces filled with zeros. A lot of programmers, however, use **strncpy** as the equivalent of the BASIC LEFT$, so that the number n is less than the number of characters in the string to be copied. Now when this is so, the correct number of characters is copied, but this does not include the terminating zero, so that the copy is not now a true string. If you have a listing of a **strncpy** function, then it's useful to make a new version in which n-1 characters are copied, and the terminating zero is inserted. It's possible, of course, that your function library may have such a function, or that its version of **strncpy** includes this action.

Another useful pair of string actions are **strcat** and **strncat**. These concatenate one string to another, the action that in most versions of BASIC is accomplished by the + operator. Once again, you have to be certain that the string to which the addition is made has been dimensioned is long enough. The number-limited version **strncat** will add only n characters of the second string, and will put in the zero terminator correctly. Your library should also include **strlen** for finding the number of characters in a string, meaning the characters before the zero. Many libraries will also include a number of functions that carry out the actions of testing a string for the occurrence of given characters.

The simplest version is **strchr(string,c)** which looks in a string for a given character and returns with a pointer to the position of the first occurrence of this character in the string. This carries out an interesting form of RIGHT$ action, because assigning this pointer to a variable name and printing as a string will give the string from the character position onwards. The pointer value will be NULL if the string does not contain the character. You can use this function to find the end of any string by using **strchr(string,0)**. Another character-in-string function will find the *last* occurrence of a given character in a string, and other functions are concerned with finding if any characters in one string occur in another string. The larger libraries also provide for finding one string inside another.

Other functions

The other functions that you are likely to find in a C library cannot be put so conveniently into categories. Much depends on the capabilities of your C compiler. If your compiler is a simple integer-only type, then the standard functions of a large C library such as trigonometric functions, logarithms, square roots and all manner of floating-point arithmetic actions are of no use, and will be absent. Even if your compiler implements floating point numbers (and by no means all do even in the higher price range) the library may not contain as many functions of this type as you might like. A lot of programmers make a fashionable point of sneering at BASIC, but most varieties of BASIC contain an excellent range of trigonometrical and other mathematical functions, as you might expect of a language derived from FORTRAN. You must remember that C was originally designed for the benefit of systems programmers, programmers for programmers if you like, with no particular need for functions of that type, so that a lot of versions of C were integer-only.

Even a limited form of C, however, will have the **atoi** function that carries out the equivalent of the BASIC VAL action. The name is a shortened version of ASCII to integer, and the argument of **atoi** is a string pointer. If the string starts with a digit, or a sign character (+ or -) then the digits of the string will be converted to integer form until the first non-digit character, which might be the terminating zero of the string, is found. As you might expect, there is no checking to find if the integer will overflow, but it is not difficult to incorporate this in the routine that uses the function. The presence of **atoi** allows the entry of numbers in string form and subsequent conversion just as we do in BASIC, and is one way of avoiding the use of **scanf**. If your compiler supports floating point numbers, then the library will include the function **atof** which carries out the conversion of a string into a floating point number. You should also find the opposite conversions available, but this is much less common in small libraries.

What else you get depends very much on your compiler and your library size. It's reasonable to expect some machine-access functions, the equivalent of the BASIC PEEK and POKE, and possibly some port actions. You can expect functions **abs(n)** and **sign(n)** which provide the absolute value and the sign value respectively for a number. These correspond very closely to their BASIC equivalents. Depending on your operating system, you may get more advanced disk filing functions, include all these that are needed for random-access filing. The larger libraries, however, will incorporate much more complex functions, such as searching and sorting, the manipulation of linked lists and the memory that they need.

Chapter 12
Machine code

Normally when you program in a high level language, you try to forget about machine code. For some applications this is perfectly possible but for the kind of programs that C programmers write, the action of the machine is very important. You might, for example, wish to carry out sideways scrolling for a spreadsheet, or up/down scrolling for a word-processor, and these actions are machine-specific. At some stage, then, it's important to be able to insert some machine code directly into the compiler so that it is put unchanged into the memory of the machine.

Now this isn't a standard part of the C language, because when C was devised the operating system could cope with the requirements of systems programmers at the time – and spreadsheets hadn't been invented! If, as is likely, you aren't running under UNIX, then your compiler should have made some provision for this problem. Obviously since there is no standard to follow here, different compilers will follow different routes, but a good model is the **inline** statement of the Hisoft compiler for Amstrad machines. The syntax is that the reserved word **inline** is followed by an ordinary opening bracket, then by the bytes of machine code separated by commas.

The end of the machine code is signalled by a closing bracket. If, as is usual, this inline machine code is placed following a function header, then the function action will be the execution of the machine code. The ability to place machine-specific code into a program in this way solves a lot of problems which otherwise would severely restrict the range of applications for programs written in C. Obviously, the use of such inline code makes the program less universally suitable, but if all the inline sections are kept within functions, then the change from one machine to another can be done by rewriting these functions alone, leaving the remainder of the program intact.

Implementations of C

By now, there are a bewildering number of compilers and interpreters for C, both in 8-bit and in 16-bit form. Some will run on specific machines some under CP/M or CP/M-86, others under MS-DOS and some under UNIX if you have access to a computer that can run UNIX. Until the memory size of computers increases considerably with no increase in costs, however, the prospect of low-price UNIX machines seems as remote as it ever was, and most C programmers for small machines will be using one of the other main operating systems. Obviously, the operating system that your computer uses automatically restricts your choice of C, but you will still have the choice of several interpreters and compilers. Unless you are in the business of teaching C to others, it's not really likely that you will want to use an interpreter.

One of the main advantages of using C is that a good C compiler running under a standard operating system can provide programs that are easily transferred from one computer to another. By using an interpreter, you confine your programs to machines that are equipped with the same interpreter. Even some compiled versions of C give a code that is very close to being interpreted. Such versions allow the recording of the source program, and this is compiled each time the program is to be run. It's usual, however, to allow a fully compiled option that will run on any machine, but this will normally require considerably more memory than is needed for the part compiled, part interpreted form.

The larger compilers consist of a text editor for creating the source program that is stored on disk, and a separate compiler that will read the source code and compile, putting the compiled code on to disk directly. This compiled code should then run on any machine that uses the same operating system and can read the disks.

In general, you need to know what you are likely to want when you buy a compiler, so that if you are learning C it's a good idea to learn with the lowest cost compiler you can buy, and then move to more exotic and more expensive software later. The features of the larger compiler that you pay so much more for are likely to be features that you don't need at all when you are learning. The sort of features I mean are things like compiling a set of source code files into one compiled program, or compiling a set of compiled codes into one program. These are features that you need when you are writing very long and involved programs, and you are quite certain not to be doing this at the learner stage.

There are many other features of larger compilers that are useful mainly to the programmer who makes a living from writing programs in C, and even for such a programmer, not all of these features are used all the time. It's true of practically any programming language that some 20% of the language is used for 80% of the time, and the ratio for compiler features may well be even larger. Start small, and grow when you have to. That way you'll waste less time wondering just how you can use all these features that you paid so much for.

Errors

BASIC interpreters generally have mastered the art of meaningful error messages. The trouble with a compiled language is that there are some errors that can be picked up by the compiler, some that are picked up during running, and some that you learn about when your keyboard goes dead and nothing appears on the screen. This is not a problem that is peculiar to C, it is a problem of any compiled language. When an interpreted language program is running each instruction is run as a separate effort, and an error can be signalled at the time of running and located to one particular statement. This is much less easy with a compiled language, because many errors can be picked up only several statements later during compilation. Errors that occur during running mean that the error will have to be corrected and the program re-compiled before another run can be made. Errors that occur in valid statements and will run but that nevertheless corrupt the memory will cause a program crash that usually requires rebooting the entire system.

The errors that are picked up by the compiler result in error messages, but the purpose of these messages is not always clear to you until you have had some experience with C. One that always causes confusion takes the form 'not an lvalue', because this is a concept that you don't usually have to worry about in BASIC. Without going into the complications of exact definition, an lvalue is something that you would expect to see on the left hand side of a comparison, like a=n-2. The error message means that you have tried to carry out an assignment to something that can't be assigned in this way. The usual cause of the message is that you have forgotten that the name of a string is a pointer, and have tried to assign a string name. Another cause of the lvalue error is trying to use a function name in an assignment, but the string name is by far the more usual problem if you have previously programmed in BASIC.

Another similar type of message is of the form 'primary expected'. A primary means anything that can be represented by a name, a variable, a member of a structure, an array or whatever. This makes it look as if the error message should be a clear one, but it's not always apparent why it has been delivered. You will often need to scan all the source program just prior to the appearance of the error message to find what has gone wrong. The point is that the compiler has issued the message under the impression that a variable name is missing in a function. The actual error may not be obviously connected with the message (like a reminder line with no */ at the end), but it should be locatable if you go through the code trying to think what the compiler will be making of it.

Other error messages tend to be more straightforward, but remember that your compiler may have a lot of error messages that are peculiar to that compiler. For example, if your compiler is integer only, any attempt to declare a number as **float** must cause an error message. Other compiler or machine limitations, such as on the number of **case** statements following a **switch**, the number of global variables, too many variable types, stack full, are all specific to your machine and compiler and are not really related to C as such. You can expect things to be quite different on another machine and another compiler— but the lvalue and primary message will always appear until you learn by experience how to avoid them!

Pointers to functions

Everything in C has a pointer, and a function is no exception, since the code of a function will be stored starting at some address in the memory, and this address number is a pointer to the function. The two things that C allows you to do with a function are to call it, and to find its pointer so that it can be called in this way. When you first start to learn programming in C , pointers to functions are not exactly the most urgent item on the agenda but there may come a time when the use of a function pointer is the easiest way out of a problem, allowing you to have a stage in a program at which one function can be selected rather than another. A very simple example of the use of a pointer to a function is shown in Figure 12.1. The example is a *very* elementary one, which means that what it does could very easily be done by simpler methods. The point, however, is that it's only by looking at fairly simple examples that you can disentangle the important features from all the rest. In this example, the program is provided with a word which is typed partly in upper-case and partly in lower-case, and the aim is to reverse the case of each letter by calling appropriate functions.

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

12.1 Passing pointers to functions. This allows functions to be chosen by using their pointers, as this simple program illustrates.

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

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

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

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

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

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

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

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

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

to ensure the return of a suitably changed character. The effect of all this will be to pass the pointer to one of the functions **up** or **down**, and execute the action of that function from within **redo**. It's not exactly the kind of thing that you can do in BASIC, and it's one of the many features that makes C such a very powerful language for programming. The functions **up, down** are written conventionally, with **return(s)** used to return values. One oddity of my compiler was that it allowed these lines to be omitted without causing problems, but I don't recommend this on your compiler!

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

Some BASIC conversions

When you make the transition from BASIC to C you very often still continue to think in terms of BASIC for some time. This can often mean that you need to find a C equivalent of BASIC actions, and this Chapter concludes with a list of a few selected BASIC reserved words with advice on the equivalents in C. Obviously there are many varieties of BASIC, and quite a few of C, so that a lowest common denominator of actions has been selected here. I have omitted all actions that depend on floating point arithmetic, for example, because the correspondence between the BASIC reserved word and the C library function is usually very close. I have also omitted all of the actions that in BASIC would be machine-specific, such as graphics instructions. What remains is a form of basic BASIC and its C equivalent, a crutch that can be thrown away as soon as your experience of C has been enough to persuade you to make the complete conversion in thinking habits. The form of this summary is that the BASIC reserved word is given in capitals, and the comments on the C equivalent follow. In a few cases, sample functions have been shown. These are in 'skeleton' form, that is stripped of all checks so that they occupy as little space as possible.

ABS — there will usually be a function **abs(n)** in the library, even for a small integer-only C . The function returns the absolute value of **n**, stripped of any negative sign.

ASC — not needed as a function because of the way that strings are used. If s is the name of a string in C, then ***s** will be the ASCII code for the first character, which is what the ASC(A$) action is in BASIC.

CHR$ — carried out by the **putchar()** function, or as part of **printf.**

CLS— usually dealt with by **printf("\f")**, usually as the first part of a larger **printf** statement.

DATA — not used because the action is that of initialising a variable or array and this can be done as part of the declaration providing that the array is declared as **static**.

DIM — the C equivalent is contained in the declaration of an array. Note that C requires any array to be fully declared, there is no 'implied' dimensioning of ten subscripts as there is in most versions of BASIC.

END — carried out by the closing curly bracket.

FOR — the **for** reserved word. Note that the conditions that follow **for** in C must be enclosed in brackets, and any or even all conditions can be omitted, altering the way that the loop acts.

GOSUB — action redundant, carried out by using the name of a function.

GOTO — C uses **goto label** and the label must be declared and correctly placed.

IF — use **if** with condition(s) placed within brackets.

INKEY — usually a library function such as **rawin()** in Hisoft C.

INPUT — use **scanf** for keyboard input, or **fscanf** to carry out the action of INPUT#n to read from a file.

INSTR— if the BASIC action of finding a string as distinct from a character is needed, then the routine shown in Figure 12.2 is useful. This, like its BASIC equivalent returns zero if the small string is not contained in the big one, and returns the starting position in the big string if the small string is contained. The syntax is of the **form n=instr(big,small).**

```
int instr(big,small)
char *big,*small;
{
 static int length;
 char *s;
 length=strlen(small);
 s=big;
 do
  {
  if (!strncmp(big,small,length)) return(big-s);
  }
 while(*++big);
return (0);
}
```

12.2 A function for the BASIC INSTR action.

LEFT$ — the function **left(s,n)** in Figure 12.3 will return a string that consists of the left **n characters of string s,** and this returned string can be asigned using **strcpy**.

```
char *left(s,n)
char *s;
int n;
{
 char *p;
 p=s;
 while(n--)p++;
*p=0;
return (s);
}
main()
{
char s[40],str[40];
scanf("\n%s",s);
printf("\n%s",left(s,3));
}
```

12.3 The action of LEFT$ carried out in a C function.

LEN — is carried out using the library function **strlen.**

MID$ — can be dealt with using the function shown in Figure 12.4. This operates with string s and copies **len** characters starting at position **pos,** numbered conventionally with the first character 1 (not 0).

```
char *mid(s,pos,len)
char *s;
int pos,len;
{
 char *p;
 while (pos-1)
 {
  s++;pos--;
 }
 p=s;
 while(len)
 {
  p++;len--;
 }
 *p=0;
 return(s);
}
main()
{
char s[40];
scanf("\n%s",s);
printf("\n%s",mid(s,3,4));
}
```

12.4 Obtaining the MID$ action by using pointers.

NEXT— not needed because the end of a **for** loop is marked either by a semicolon or a curly bracket.

ON GOSUB — use a **switch** statement, but don't forget the break statement following each **switch** line.

PRINT — use function **printf**, which is much more flexible.

PRINTAT — no simple general routine, because this can be very strongly machine-specific. There may be an equivalent in your library.

READ — see DATA.

REM — use /*   */

RESTORE — not needed because of initialisation action.

RETURN — not specifically needed because a function returns when the curly bracket at the end of the function is read. Otherwise use **return()** or simple **return**.

RIGHT$ — see the routine in Figure 12.5.

```
char *right(s,n)
char *s;
int n;
{
 while(*++s);
 while(n--) s--;
 return(s);
}
main()
{
char s[40];
scanf("\n%s",s);
printf("\n%s",right(s,3));
}
```

12.5 The RIGHT$ action in C.

SGN — most C libraries include the function **sign().**

SPC — use the formatting action of **printf.**

STR$ — use **sprintf** to place a number into a string.

STRING$ — see the function **filstr** in Figure 12.6. This assigns to string s number **n** identical characters **c**. The characters can be in ASCII code form or as characters using single quotes.

```
char *filstr(s,n,c)
char *s,c;
int n;
{
 while(n--) *s++=c;
 *s=0;
 return (s);
}
main()
{
char s[20];
filstr(s,10,'*');
printf("\n%s",s);
}
```

12.6 A filstr function that carries out the BASIC STRING$(n,'c') action.

TAB — use the formatting action of **printf.**

VAL — obtained as part of the action of **sscanf**, using a number specifier to read from a string.

Appendix A
Ones Complement

The ones complement of a binary number is the result of inverting each digit in the number. Inversion means that each 1 is replaced by a 0 and each 0 by a 1.

For example, the single character number denary 59 is in binary:

```
    0011 1011
```

and this complements to:

```
    1100 0100
```

which in denary is 196.

For integers using two bytes, the action is identical. The number 1706 is in binary:

```
    0000 0110 1010 1010
```

which complements to:

```
    1111 1001 0101 0101
```

giving (unsigned) denary 63829.

Note that if signed denary numbers are used, these results may be shown in negative form. For a short integer of two bytes, any number greater than 32767 will appear as a negative number obtained by subtracting the result number from 65536. The number 63829, for example, will appear as -1707.

Appendix B Other binary number operators

NOTE: These operators apply to types char and int (long or short) only.

1. The shift operators

The shift operators are **>>** and **<<**, each of which can be used along with two numbers, the number that is being shifted and the shift number. If the shift number is not stated, the default is unity. The syntax is of the form **x>>y** where **x** is the number operated on, and **y** is the number of binary places shifted. The action is most easily illustrated with a single-byte (char) number.

For example, the expression 45<<2 means a two position left shift of the number 45 denary. The action in binary can be seen by writing 45 in binary as:

```
    0010 1101
```

and then shifting each bit one place left. The lefthand bit disappears, and a zero is placed at the right hand side, giving:

```
    0101 1010   (denary 90)
```

and a second shift on this result gives

```
    1011 0100   (denary 180)
```

The left shift action has the effect of multiplying numbers by 2, but this will not always appear to be true if the most significant bit (left hand side) of an integer is changed, because this bit is used as a sign indicator. Even if you work with unsigned numbers incorrect answers can be obtained because of overflow. For example, the integer number:

```
0100 0110 1101 1001  (denary 18137)
```

shifted three times left gives :

```
0011 0110 1100 1000  (denary 14024)
```

because of the overflow from the left hand side.

Similarly, a right shift is equivalent to an integer division by 2, but the displayed result can seem incorrect if there is a change in the most significant bit (left hand side), or if the shifting leaves the number all-zero. For example, the short integer that in signed form is -24 is in binary:

```
1111 1111 1110 1000
```

and the three-place right shift (divide by eight) gives:

```
0001 1111 1111 1101
```

which is denary 8189. This problem does not appear if integer numbers are displayed in unsigned form, when the two numbers are shown as 65512 and 8189, because in this case the relationship 65512/8=8819 is obvious.

2. The &, |, ^ operators

These operators apply to pairs of numbers, and their actions can be seen when the numbers are put into binary form. The basic actions are described in the tables and summaries of Figure 5.6.

The & action will produce a 1 bit only when both bits being compared are 1. The bits being compared are the corresponding bits in each byte. For example, if the integer 23176 is anded with 10712, then in binary this is:

```
0101 1100 1000 1000  (23176)
0010 1001 1101 1000  (10712)
-------------------------------
0000 1000 1000 1000  (2184)
```

The & action is used in masking, for example to isolate a group of bits for test purposes. For example, masking with 0000 1111 (denary 15) will isolate the lower four bits of a number. The number 74 denary is:

```
0010 1010  and masking with 0000 1111 gives:
0010 1010
0000 1111
_________
0000 1010
```

producing the lower half-byte or nibble. The OR action of the | operator produces a 1 if either or both bits being compared is 1 - conversely it produces a zero only if both bits are zero. For example, the integer result of 17025|51239 is shown as:

```
0100 0010 1000 0001 (17025)
1100 1000 0010 0111 (51239)
___________________________
1100 1010 1010 0111 (51879)
```

This can be used in setting a particular bit. For example, the sixth bit (right hand bit is zero) can be set by ORing with 0100 0000:

```
0011 0100
0100 0000
_________
0111 0100   sixth bit set.
```

The XOR action of ^ will give a 1 only when one of the bits being compared is 1 and the other is zero. For example:

```
0000 0111 1011 0000 (1968 denary)
0111 0110 1100 1111 (30415)
_____________________
0111 0001 0111 1111 (29055)
```

The applications to coding and decoding have been covered in the text of Chapter 5.

Index

C is the computer programming language of the eighties.

C is the language of the acclaimed UNIX™ operating system for 8, 16 and 32-bit machines, and there are versions of it available for almost all makes of home computer. There are more compilers for C than for any other language.

C for Beginners provides you with a solid grounding in this increasingly popular language. Data-types, assignments, loops, arrays, sorting and searching, pointers, graphics, sound, string-handling . . . Ian Sinclair's authoritative text covers all this and much more. The book contains routines specifically designed for use with **Hisoft C** for the Amstrad, but – because of C's portability – these will be compatible with other versions of the language, including Lattice.

C for Beginners is a complete guide to one of the most versatile computer languages yet devised. It is essential reading whether you're a budding systems programmer or just want to enhance your programming knowledge. In fact, it is essential reading for anyone who's interested in computers at all.

C for yourself.

SUITABLE FOR AMSTRAD 1512

£10.95

Melbourne House Publishers

ISBN 0-86161-206-X

9 780861 612062

C FOR BEGINNERS

Ian Sinclair

M