Introduction to

C Programming

Rob Miles
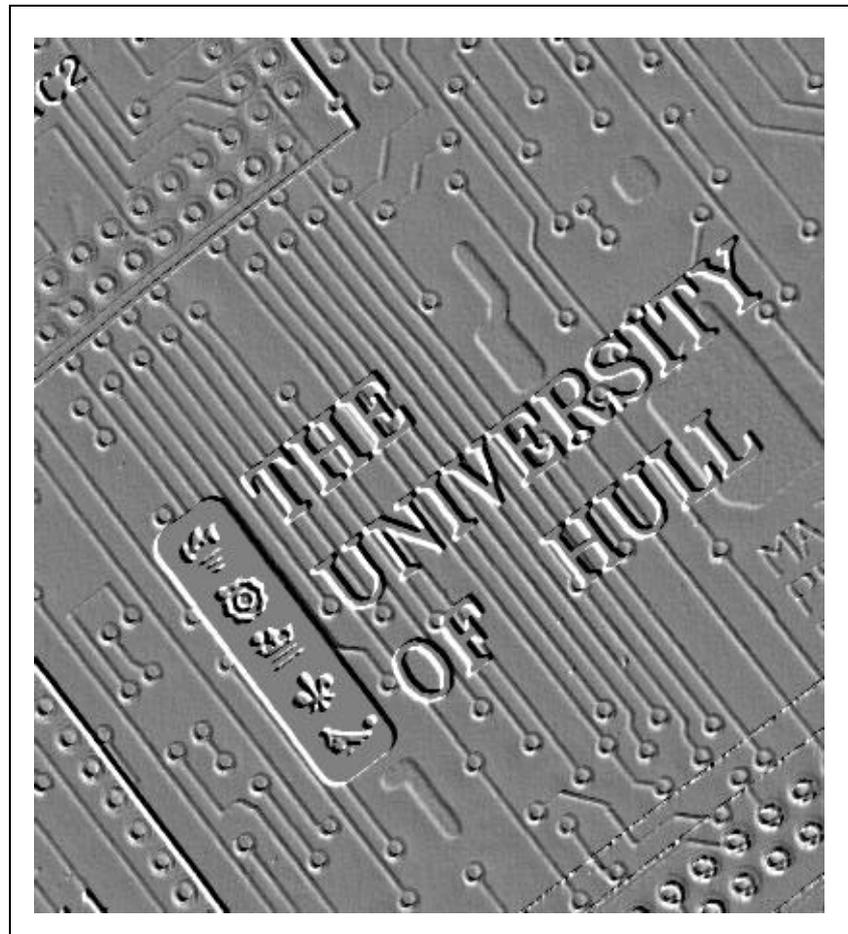
Department of Computer Science

Contents

Writing a Program 24

Functions 37

Arrays 46

Switching 53

Strings 55

The author can be contacted at: The Department of Computer Science, The University of Hull,
HULL, HU6 7RX

r.s.miles@dcs.hull.ac.uk

Computers

An Introduction to Computers

Qn: Why does a bee hum?
Ans: Because it doesn't know the words!

One way of describing a computer is as an electric box which humms. This, whilst technically correct, can lead to significant amounts of confusion, particularly amongst those who then try to program a fridge. A better way is to describe it as:

A device which processes information according to instructions it has been given.

This general definition rules out fridges but is not exhaustive. However for our purposes it will do. The instructions you give to the computer are often called a program. The business of using a computer is often called programming. This is **not** what most people do with computers. Most users do not write programs, instead they talk to programs written by other people. We must therefore make a distinction between users and programmers. A user has a job which he or she finds easier to do on a computer running the appropriate program. A programmer has a masochistic desire to tinker with the innards of the machine. One of the golden rules is that you never write your own program if there is already one available, i.e. a keen desire to process words with a computer should not result in you writing a word processor!

However, because you will often want to do things with computers which have not been done before, and further because there are people willing to pay you to do it, we are going to learn how to program as well as use a computer.

Before we can look at the fun packed business of programming though it is worth looking at some computer terminology:

Hardware and Software

If you ever buy a computer you are not just getting a box which humms. The box, to be useful, must also have sufficient built in intelligence to understand simple commands to do things. At this point we must draw a distinction between the software of a computer system and the hardware.

Hardware is the physical side of the system. Essentially if you can kick it, and it stops working when immersed in a bucket of water, it is hardware. Hardware is the impressive pile of lights and switches in the corner....

Software is what makes the machine tick. If a computer has a soul it keeps it in its software. Software uses the physical ability of the hardware, which can run programs, do something useful. It is called software because it has no physical

existence and it is comparatively easy to change. Software is the voice which says "Computer Running" in a Star Trek film.

All computers are sold with some software. Without it they would just be a novel and highly expensive heating system. The software which comes with a computer is often called its Operating System. The Operating System makes the machine usable. It looks after all the information held on the computer and provides lots of commands to allow you to manage things. It also lets you run programs, ones you have written and ones from other people. You will have to learn to talk to an operating system so that you can create your C programs and get them to go.

We are going to use an operating system called MS-DOS. Later we will be using UNIX.

Data and Information

People use the words data and information interchangeably. They seem to think that one means the other. I regard data and information as two different things:

Data is the collection of ons and offs which computers store and manipulate.

Information is the interpretation of the data by people to mean something. Strictly speaking computers process data, humans work on information. An example, the computer holds the bit pattern:

11111111 11111111 11111111 00000000

However you could regard this as meaning:

"you are 256 pounds overdrawn at the bank"

or

"you are 256 feet below the surface of the ground"

or

"eight of the thirty two light switches are off"

The transition from data to information is usually made when the human reads the output. So why am I being so pedantic? Because it is vital to remember that a computer does not "know" what the data it is processing actually means. As far as it is concerned data is just patterns of bits, it is you who gives meaning to these patterns. Remember this when you get a bank statement which says that you have £8,388,608!

Data Processing

Computers are data processors. Information is fed into them, they do something with it, and then generate further information. A computer program tells the computer what to do with the information coming in. A computer works on data in the same way that a sausage machine works on meat, something is put in one end, some processing is performed, and something comes out of the other end:
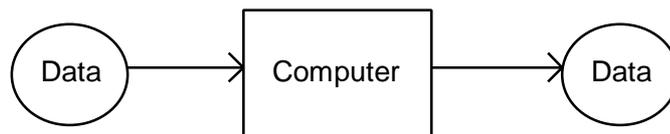
This makes a computer a very good "mistake amplifier", as well as a useful thing to blame.....

A program is unaware of the data it is processing in the same way that a sausage machine is unaware of what meat is. Put a bicycle into a sausage machine and it will try to make sausages out of it. Put duff data into a computer and it will do equally useless things. It is only us people who actually ascribe meaning to data (see above), as far a the computer is concerned it is just stuff coming in which has to be manipulated in some way.

A computer program is just a sequence of instructions which tell a computer what to do with the data coming in, and what form the data sent out will have.

Note that the data processing side of computers, which you might think is entirely reading and writing numbers, is much more than that, examples of typical data processing applications are:

Digital Watch : A micro-computer in your watch is taking pulses from a crystal and requests from buttons, processing this data and producing a display which tells you the time.

Car : A micro-computer in the engine is taking information from sensors telling it the current engine speed, road speed, oxygen content of the air, setting of the accelerator etc and producing voltages out which control the setting of the carburettor, timing of the spark etc, to optimise the performance of the engine.

CD Player : A computer is taking a signal from the disk and converting it into the sound that you want to hear. At the same time it is keeping the laser head precisely positioned and also monitoring all the buttons in case you want to select another part of the disk.

Note that some of these data processing applications are merely applying technology to existing devices to improve the way they work. However one, the CD player, could not be made to work without the built-in data processing ability.

Most reasonably complex devices contain data processing components to optimise their performance and some exist only because we can build in intelligence. It is into this world that we, as software writers are moving. It is important to think of business of data processing as much more than working out the company payroll, reading in numbers and printing out results. These are the traditional uses of computers.

Note that this "raises the stakes" in that the consequences of software failing could be very damaging.

As engineers it is inevitable that a great deal of our time will be spent fitting data processing components into other devices to drive them. You will not press a switch to make something work, you will press a switch to tell a computer to make it work. These embedded systems will make computer users of everybody, and we will have to make sure that they are not even aware that there is a computer in there!

Programming Languages

What is Programming?

I tell people I am a "Software Engineer".

Programming is a black art. It is the kind of thing that you grudgingly admit to doing, at night, with the blinds drawn and nobody watching. Tell people that you program computers and you will get one of the following responses:

1. A blank stare.

2. "That's interesting", followed by a long description of the double glazing that they have just had fitted.

3. "My younger brother has a Sinclair Spectrum. He's a programmer as well."

4. A look which indicates that you can't be a very good one as they all drive Ferraris and tap into the Bank of England at will.

Programming is defined by most people as *earning huge sums of money doing something which nobody can understand.*

Programming is defined by me as *deriving and expressing a solution to a given problem in a form which a computer system can understand and execute.*

One or two things fall out of this definition:

? You need to be able to solve the problem yourself before you can write a program to do it.

? The computer has to be made to understand what you are trying to tell it to do.

We are back to the term "Software Engineer" again!

I like to think of a programmer as a bit like a plumber! A plumber will arrive at a job with a big bag of tools and spare parts. Having looked at it for a while, tut tutting, he will open his bag and produce various tools and parts, fit them all together and solve your problem. Programming is just like this. You are given a problem to solve. You have at your disposal a big bag of tricks, in this case a programming language. You look at the problem for a while and work out how to solve it and then fit the bits of the language together to solve the problem you have got. The art of programming is knowing which bits you need to take out of your bag of tricks to solve each part of the problem.

From Problem to Program

*Programming is **not** about mathematics, it is about organisation.*

The art of taking a problem and breaking it down into a set of instructions you can give a computer is the interesting part of programming. Unfortunately it is also the most difficult part of programming as well. If you think that learning to program is simply a matter of learning a programming language you are very wrong. In fact if you think that programming is simply a matter of coming up with a program which solves a problem you are equally wrong!

There are many things you must consider when writing a program; not all of them are directly related to the problem in hand. I am going to start on the basis that you are writing your programs for a customer. He or she has problem and would like you to write a program to solve it. We shall assume that the customer knows even less about computers than we do!

Initially we are not even going to talk about the programming language, type of computer or anything like that, we are simply going to make sure that we know what the customer wants. Coming up with a perfect solution to a problem the customer has not got is something which happens surprisingly often in the real world.

The worst thing you can say to a customer is "I can do that". Instead you should think "Is that what the customer wants?".

This is almost a kind of self discipline. Programmers pride themselves on their ability to come up with solutions, so as soon as they are given a problem they immediately start thinking of ways to solve it, this almost a reflex action. What you should do is think "Do I really understand what the problem is?". Before you solve a problem you should make sure that you have a watertight definition of what the problem is, which both you and the customer agree on. In the real world this is often called a Functional Design Specification or FDS. This tells you exactly what the customer wants. Both you and the customer sign it, and the bottom line is that if you provide a system which behaves according to the design specification the customer must pay you. Once you have got your design specification, then you can think about ways of solving the problem.

You might think that this is not necessary if you are writing a program for yourself; there is no customer to satisfy. This is not true. Writing an FDS forces you to think about your problem at a very detailed level.

A Simple Problem

Consider the scenario; You are sitting in your favourite chair in the pub contemplating the universe when you are interrupted in your reverie by a friend of yours who sells double glazing for a living. He knows you are a programmer of sorts and would like your help in solving a problem which he has:

He has just started making his own window units and is looking for a program which will do the costing of the materials for him. He wants to just enter the dimensions of the window and then get a print out of the cost to make the window, in terms of the amount of wood and glass required.

"This looks like a nice little earner" you think, and once you have agreed a price you start work. The first thing you need to do is find out exactly what the customer wants you to do...

Specifying the Problem

When considering how to write the specification of a system there are three important things :

- ? What information flows into the system.
- ? What flows out of the system.
- ? What the system does with the information.

There are lots of ways of representing this information in the form of diagrams, for now we will stick with written text when specifying each of the stages:

Information going in

In the case of our immortal double glazing problem we can describe the information as:

- ? The width of a window.
- ? The height of the window.

Information coming out

The information that our customer wants to see is :

- ? the area of glass required for the window
- ? the length of wood required to build a frame.

What the program actually does

The program can derive the two values according to the following equations :

glass area = width of window * height of window
wood length = (width of window + height of window) * 2

Putting in more detail

We now have a fairly good understanding of what our program is going to do for us. Being sensible and far thinking people we do not stop here, we now have to worry about how our program will decide when the information coming in is actually valid.

This must be done in conjunction with the customer, he or she must understand that if information is given which fits within the range specified, your program will regard the data as valid and act accordingly.

In the case of the above we could therefore expand the definition of data coming in as :

- ? The width of the window, in metres and being a value between 0.5 Metres and 3.5 metres inclusive.
- ? The height of the window, in metres and being a value between 0.5 metres and 2.0 metres inclusive.

Note that we have also added units to our description, this is very important - perhaps our customer buys wood from a supplier who sells by the foot, in which case our output description should read :

- ? The area of glass required for the window, in square metres.
- ? The length of wood required for the frame, given in feet using the conversion factor of 3.25 feet per metre.

*Note that both you and the customer **must** understand the document!*

Having written this all up in a form that both you and the customer can understand, we must then both sign the completed specification, and work can commence. In a real world you would now create a procedure which will allow you to prove that the program works, you could for example say :

If I give the above program the inputs 2 metres high and 1 metre wide the program should print out : 2 square metres of glass and 9.75 feet of wood.

The test procedure which is designed for a proper project should test out all possible states within the program, including the all important error conditions.

In a large system the person writing the program may have to create a test harness which is fitted around the program and will allow it to be tested. Both the customer and the supplier should agree on the number and type of the tests to be performed and then sign a document describing these.

Better yet, set up a phased payment system so that you get some money as the system is developed.

At this point the supplier knows that if a system is created which will pass all the tests the customer has no option but to pay him for the work! Note also that because the design and test procedures have been frozen, there is no ambiguity which can lead to the customer requesting changes to the work although of course this can still happen!

Note also in a "proper" system the customer will expect to be consulted as to how the program will interact with the user, sometimes even down to the colour of the letters on the display! Remember that one of the most dangerous things that a programmer can think is "This is what he wants"! The precise interaction with the user - what the program does when an error is encountered, how the information is presented etc., is something which the customer is guaranteed to have strong opinions about. Ideally all this information should be put into the specification, which should include layouts of the screens and details of which keys should be pressed at each stage.

Fact: If you expect to derive the specification as the project goes on either you will fail to do the job, or you will end up performing five times the work!

If this seems that you are getting the customer to help you write the program then you are exactly right! Your customer may have expected you to take the description of the problem and go into your back room - to emerge later with the perfect solution to the problem. This is not going to happen. What will happen is that you will come up with something which is about 60% right. The customer will tell you which bits look OK and which bits need to be changed. You then go back into your back room, muttering under your breath, and emerge with another system to be approved. Again, Robert's law says that 60% of the duff 40% will now be OK, so you accept changes for the last little bit and again retreat to your keyboard....

The customer thinks that this is great, reminiscent of a Saville Row tailor who produces the perfect fit after numerous alterations. All the customer does is look at something, suggests changes and then wait for the next version to find something wrong with.....

Fact: More implementations fail because of inadequate specification than for any other reason!

If your insisting on a cast iron specification forces the customer to think about exactly what the system is supposed to do and how it will work, all to the better. The customer may well say "But I am paying you to be the computer expert, I know nothing about these machines". This is no excuse. Explain the benefits of "Right First Time" technology and if that doesn't work produce a revolver and force the issue!

Again, if I could underline in red I would : All the above apply if you are writing the program for yourself. You are your own worst customer!

You may think that I am labouring a point here, the kind of simple systems we are going to create as we learn to program are going to be so trivial that the above techniques are far too long winded. You are wrong. One very good reason for doing this kind of thing is that it gets most of the program written for you - often with the help of the customer. When we start with our double glazing program we now know that we have to :

read in the width
verify the value
read in the height
verify the value
calculate width times height and print it
calculate (width + height) * 2 * 3.35 and print it

The programming portion of the job is now simply converting the above description into a language which can be used in a computer.......

Programming Languages

You might ask the question "Why do we need programming languages, why can we not use something like English?" There are two answers to this one:

1. Computers are too stupid to understand English.

2. English would make a lousy programming language.

Please note that this does not imply that tape worms would make good programmers!

To take the first point. We cannot make very clever computers at the moment. Computers are made clever by putting software into them, and there are limits to the size of program that we can create and the speed at which it can talk to us. At the moment, by using the most advanced software and hardware, we can make computers which are about as clever as a tape worm. Tape worms do not speak very good English, therefore we cannot make a computer which can understand English. The best we can do is get a computer to make sense of a very limited language which we use to tell it what to do.

Time Files like an Arrow. Fruit Flies like a Banana!

To take the second point. English as a language is packed full of ambiguities. It is very hard to express something in an unambiguous way using English, if you do not believe me, ask any lawyer!

Programming languages get around both of these problems. They are simple enough to be made sense of by computer programs and they reduce ambiguity. There are very many different programming languages around, you will need to know more than one if you are to be a good programmer.

C

A look at C

There are literally hundreds of programming languages around, you will need to know at least 3!

We are going to learn a language called C. C is a very flexible and powerful programming language originally designed in the early 1970s. It is famous as the language the UNIX operating system was written in, and was specially designed for this. However its use has now spread way beyond that field and it is currently very popular.

C is a *professional* language. So what do I mean by that? Consider the chain saw. If I, Rob Miles, want to use a chain saw I will hire one from a shop. As I am not an experienced chain saw user I would expect it to come with lots of built in safety features such as guards and automatic cut outs. These will make me much safer with the thing but will probably limit the usefulness of the tool, i.e. because of all the safety stuff I might not be able to cut down certain kinds of tree. If I was a real lumberjack I would go out and buy a professional chain saw which has no safety features whatsoever but can be used to cut down most anything. If I make a mistake with the professional tool I could quite easily lose my leg, something the amateur machine would not let happen.

In programming terms this means is that C lacks some safety features provided by other programming languages. This makes the language much more flexible.

However, if I do something stupid C will not stop me, so I have a much greater chance of crashing the computer with a C program than I do with a safer language.

This is not something to worry about, you should always work on the basis that any computer will tolerate no errors on my part and anything that I do which is stupid will always cause a disaster!

Making C Run

You actually write the program using some form of text editor - which may be part of the compiling and linking system.

C is usually a *compiled* programming language. The computer cannot understand the language directly, so a program called a compiler converts the C into the machine code instructions which do the job. Actually getting a program to run is however a two stage process. First you show your program, often called the *source*, to the compiler. If the compiler gives it the thumbs up you then perform a process called linking.

Linking is when all the various portions of your program are brought together to form something which can be run. You might ask "Why to we link things? - the compiler has created a machine code version of my program, can't I just run that?". The reason that we have the additional linking process is that it allows us to reuse standard pieces of code from a library. Many things your program will do are common to lots of other programs, for example you will want to read information from the keyboard and you will want to send information to the display. Rather than compile the program code which does this every time you compile your program, a much more efficient way is to put a compiled version of this code into a library. Your program just contains a reference to the particular function you want to use, the linker then loads the relevant part from the library when it creates your program.

It is possible to get the compiler to give you warnings in this case.

Note that a side effect of this is that if you refer to a function which does not exist, the compiler will not mind particularly - but the linker will not find the item in its library, and thus give you an error.

Once the linker has finished you are left with a free standing file which is your program. If you run this your program gets control!

Creating C Programs

The actual business of constructing and compiling the depends on the computer you are using and the particular version of C. We will look at the business of producing your program in the laboratory section of this course. Initially it is best if we just work through your programs on paper. I reckon that you write programs best when you are not sitting at the computer, i.e. the best approach is to write (or at least map out) your solution on paper a long way away from the machine. Once you are sitting in front of the keyboard there is a great temptation to start pressing keys and typing something in which might work. This is not good technique. You will almost certainly end up with something which almost works, which you will then spend hours fiddling with to get it going.

If you had sat down with a pencil and worked out the solution first you would probably get to a working system in around half the time. I am not impressed by hacking programmers who spend whole days at terminals fighting with enormous programs and debugging them into shape. I am impressed by someone who turns up, types in the program and makes it work first time!

What Comprises a C Program?

A program is the thing that you write to perform a particular task.

It will actually be a file of text, often called a source file. This is what the compiler acts on. A source file contains three things :

- ? instructions to the compiler
- ? information about the structures which will hold the data to be stored and manipulated.
- ? instructions which manipulate the data.

To take these in turn

Controlling the Compiler

The compiler directives are the same for all versions of C.

One of the very powerful features of C is the way in which you can change the way the compiler processes your program by including directives. There are many directives available, in a C program a directive is always preceded by the **#** character and must appear right at the beginning of a line. Directives can be used to "build in" particular values, for example constants like PI, and also allow you to change which parts of the program which the compiler works on, making it possible to use the same piece of program on several different types of computer.

Storing the Data

Programs work by processing data. The data has to be stored within the computer whilst the program processes it. All computer languages support variables of one form or another. A variable is simply a named location in which a value is held whilst the program runs. C also lets you build up *structures* which can hold more than one item, for example a single structure could hold all the information about a particular bank customer.

Describing the Solution

The actual instructions which describe your solution to the problem must also be part of your program. In the case of C a lump of program which does one particular thing is called a function.

Seasoned programmers break down a problem into a number of smaller ones and make a function for each.

A function can be very small, or very large. It can return a value which may or may not be of interest. It can have any name you like, and your program can contain as many functions as you see fit. One function may refer to others. The C language also has a large number of *function libraries* available which you can use. These save you from "re-inventing the wheel" each time you write a program.

Within the function there will be a number of statements. A statement is an instruction to perform one particular operation, for example add two numbers together and store the result. The really gripping thing about programs is that a statement can decide which statement is performed next, so that your program can look at things and decide what to do.

You give a name to each function that you create, and you try to make the name of the function fit what it does, for example **menu** or **save_file**. The C language actually runs your program by looking for a function with a special name, **main**. This function is called when your program starts running, and when **main** finishes, your program ends.

The Advantages of C

The good news about C is that you can write code which runs quickly, and your program is very "close to the hardware". By that I mean that you can access low level facilities in your computer quite easily, without the compiler or run time system stopping you from doing something potentially dangerous.

The use of compiler directives to the pre-processor make it possible to produce a single version of a program which can be compiled on several different types of computer. In this sense C is said to be very portable. The function libraries are standard for all versions of C so they can be used on all systems.

The Disadvantages of C

The disadvantages of C fall neatly from the advantages. The biggest one is that you can write C programs which can fail in very catastrophic ways. These programs will appear totally valid as far as the compiler is concerned but will not work and may even cause your computer to stop. A more picky language would probably notice that you were doing something stupid in your program and allow you to find the error before it crashed your computer! However a more picky language would probably not allow you to write the program in the first place!

Another disadvantage of C is that it allows you to write very *terse* code. You can express exactly what you want to do in very few statements. You might think that this is nice, because it makes your programs even more efficient, but it has the side effect of making them much harder to understand. At the time you write the code you know exactly what each part is supposed to do. If you come back to the program in several months you will need time to "get back inside it". If the code is written very tightly you will take much longer to do this, and other people may not be able to understand it at all! I write code which is not the most efficient possible, but is easy to understand. I am sacrificing program performance for ease of maintenance.

A First C Program

The Program Example

Perhaps the best way to start looking at C is to jump straight in with our first ever C program. Here it is:

```
#include <stdio.h>

void main ( void )
{
    float height, width, area, wood_length ;
    scanf ( "%f", &height ) ;
    scanf ( "%f", &width ) ;
    area = 2 * height * width ;
    wood_length = 2 * ( height + width ) * 3.25 ;
    printf ( "The area of glass is : %f metres. \n",
        area ) ;

    printf ( "The length of wood is : %f feet. \n",
        wood_length ) ;
}
```

You should easily work out what it does, but what do all the various bits mean?

#include

The pre-processor is the part of the compiler which actually gets your program from the file.

This is a pre-processor directive. It is not part of our program, it is an instruction to the compiler to make it do something. It tells the C compiler to **include** the contents of a file, in this case the system file **stdio.h**. The compiler knows it is a system file, and therefore must be looked for in a special place, by the fact that the name is enclosed in **<>** characters. (see later)

<stdio.h>

All versions of C have exactly the same library functions.

This is the name of the standard library definition file for all STanDard Input Output. Your program will almost certainly want to send stuff to the screen and read things from the keyboard. **stdio.h** is the name of the file in which the functions that we want to use are defined. A function is simply a chunk of program that we want to use a lot, so we stuck it in a parcel and gave it a name. The function we want to use is called **printf** (see later). To use **printf** correctly C needs to know what it looks like, i.e. what things it can work on and what value it returns. The actual code which performs the **printf** will be tied in later by the linker. Note that without the definition of what **printf** looks like the compiler makes a guess when it sees the use of it. This can lead to the call failing when the program runs, a common cause of programs crashing.

The .h potion is the language extension, which denotes an include file.

The **<>** characters around the name tell C to look in the system area for the file **stdio.h**. If I had given the name "**robsstuff.h**" instead it would tell the compiler to look in the current directory. This means that I can set up libraries of my own routines and use them in my programs, a very useful feature.

void

Some C programs have the type of int so that the main function can return a value to the operating system which runs it. We are not going to do this.

Means literally this means nothing. In this case it is referring to the function whose name follows. It tells C that this function, which could return something interesting, (see **printf**) in fact returns nothing of interest whatsoever. Why do this? Because we want to be able to handle the situation where I make a mistake and try to ascribe meaning to something which has none. If the C compiler has already been told that a given entity has no meaning it can detect my mistake and produce an error.

main

The name of the function currently being defined. The name **main** is special, in that the **main** function is actually the one which is run when your program is used. A C program is made up of a large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs. C regards the name "**main**" as a special case and will run this function first. If you forget to have a main function, or mistype the name, the compiler will give you an error.

(void)

This is a pair of brackets enclosing **void**. This may sound stupid, but actually tells the compiler that the function **main** has no parameters. A parameter to a function gives the function something to work on. When you define a function you can tell C that it works on one or more things, for example **sin(x)** could work on a floating point value of angle **x**. We will cover functions in very great detail later in this course.

{

This is a *brace*. As the name implies, braces come in packs of two, i.e. for every open brace there must be a matching close. Braces allow me to lump pieces of program together. Such a lump of program is often called a *block*. A block can contain the declaration of variable used within it, followed by a sequence of program statements which are executed in order. In this case the braces enclose the working parts of the function **main**.

When the compiler sees the matching close brace at the end it knows that it has reached the end of the function and can look for another (if any). The effects of an un-paired brace are invariably fatal....

float

Our program needs to remember certain values as it runs. Notably it will read in values for the width and height of the windows and then calculate and print values for the glass area and wood length. C calls the place where values are put variables. At the beginning of any block you can tell C that you want to reserve some space to hold some data values. Each item you can hold a particular kind of value. Essentially, C can handle three types of data, floating point numbers, integer number and characters (i.e. letters, digits and punctuation).

You declare some variables of a particular type by giving the type of the data, followed by a list of the names you want the variables to have. We will look at the precise rules which apply when you invent a variable name in more detail later, for now you can say that the variable must start with a letter and from then on only contain letters, numbers and the _ character.

height, width, area, wood_length

This is a list. A list of items in C is separated by the **,** character. In this case it is a list of variable names. Once the compiler has seen the word **float** (see above) it expecting to see the name of at least one variable which is to be created. The compiler works its way down the list, creating boxes which can hold floating point values and giving them the appropriate names. From this point on we can

refer to the above names, and the compiler will know that we are using that particular variable.

;

The semicolon marks the end of the list of variable names, and also the end of that declaration statement. All statements in C programs are separated by the **;** character, this helps to keep the compiler on the right track.

The **;** character is actually very important. It tells the compiler where a given statement ends. If the compiler does not find one of these where it expects to see one it will produce an error. You can equate these characters with the sprocket holes in film, they keep everything synchronised.

scanf

If you have been used to other programming languages you might expect that the printing and reading functions are part of the language. In C this is not the case, instead they are defined as standard functions which are part of the language specification, but not part of the language itself. Any decent book on C must have a big section on how to use the standard functions and what they are. The standard input/output library contains a number of functions for formatted data transfer, the two we are going to use are **scanf** (scan formatted) and **printf** (print formatted).

A parameter is something for a function to work on.

This line is a call to the function **scanf**. The compiler knows that this is a function call by the fact that it is followed by a parameter list. (see later) What the compiler does at this point is parcel up the parameters you have given and then call scanf, passing the parameter information on. This function is expecting a particular sequence and type of parameters.

The compiler knows what **scanf** looks like and the kind of information **scanf** uses is expecting because it has already seen the file **stdio.h**, which is where this function is defined. The compiler can now check that the way we have used **scanf** agrees with the definition. This allows valuable checking to be carried out. If the compiler had not seen this function defined before it would simply guess what it was supposed to do and probably produce a program which would not work properly.

(

This marks the start of the list of parameters to the **scanf** function. A parameter is something which a function operates on. All functions must have a parameter list, if they have no parameters the list is empty. Note that the parameters you give when you call a function must agree in number and type with those that the function expects, otherwise unpredictable things will happen. The system file **stdio.h** contains a description of what **scanf** should be given to work on. If what you supply does not agree with this the compiler will generate an error for you.

"%f",

This is the set of parameters to the call of **scanf**. scanf is short for scan formatted. The function is controlled by the format string, which is the first parameter to the function. The second and successive parameters are addresses into which **scanf** puts the values it has been told to fetch.

In C a string is given as a sequence of characters enclosed in **"** characters. You will meet the concept of delimiters regularly in C. A delimiter is a particular character which is to be used to define the limits of something. C uses different delimiters in different places, the **"** character is used to delimit a string of text. When the compiler sees the first **"** it recognises that everything up until the next **"** is a sequence of characters which we wish to use. It therefore just assembles the string until it finds another **".** The string must all appear on the same line, otherwise you will get an error.

The **scanf** function has been told to look out for certain characters in the format string and treat them as special. Such a special character is **%.** The **%** character tells **scanf** that we are giving the format of a number which is to be processed. The letter after the **%** character tells **scanf** what kind of value is being fetched, **f** means floating point. This command will cause **scanf** to look for a floating point number on the input. If it finds one it is to put the value in the address given as the next parameter.

&height

In C a function cannot change the value of a parameter which is supplied to it. This is quite a departure from other languages, which let you do this. It does make life more complicated, when you want your function to be able to change the value of something. The way that you get the effect is by cheating. Instead of passing **scanf** the variable height we have instead supplied **&height**. The **&** is very important. When the compiler sees the **&** it regards this as an instruction to create a pointer to the given variable, and then pass this pointer to **scanf**. This means that when **scanf** runs it is given a a pointer to where the result is to be placed, which is OK.

) ;

The **)** character marks the end of the list of parameters to **scanf** and the **;** then end of this statement.

scanf ("%f", &width) ;

The function of this line is identical to the one above, except that the result is placed into the variable called **width**.

area = 2 * height * width ;

This is an assignment. The assignments are the bread and butter of programming. A good proportion of your programs will be instructions to assign new values to variables, as the various results are calculated.

C uses the **=** character to make assignments happen. The first part of this statement is the name of a previously defined variable. This is followed by the **=** character which I call the gozzinta. I call it that because the value on the right gozzinta (goes into) the variable on the left. When this program runs the expression is worked out and then the result is placed in the specified variable. In this case the expression works out the total amount of glass needed. This result is then placed in the **area** variable.

wood_length = 2 * (height + width) * 3.25 ;

When I write programs I use brackets even when the compiler does not need them. This makes the program clearer.

This is an expression much like above, this time it is important that you notice the use of parenthesis to modify the order in which values are calculated in the expression. Normally C will work out expressions in the way you would expect, i.e. all multiplication and division will be done first, followed by addition and subtraction. In the above expression I wanted to do some parts first, so I did what you would do in mathematics, I put brackets around the parts to be done first.

printf

The **printf** function is the opposite of **scanf**. It takes text and values from within the program and sends it out onto the screen. Just like **scanf** it is common to all versions of C and just like **scanf** it is described in the system file **stdio.h**.

The first parameter to **printf** is the format string, which contains text, value descriptions and formatting instructions.

("The area of glass needed is : %f metres. \n",

This is another format string. The text in this message will appear as it is given, although the delimiters will not be printed. Like **scanf** the **%** tells **printf** that a value description follows, the **f** meaning that a floating point value is to be printed. You can get quite clever with place markers, using them to specify exactly how many spaces to fill with the value and, with real numbers, how many places after the decimal point to print, for example :

Format	Command
%d	I want you to print the value of a signed, decimal integer
%c	I want you to print the character which responds to this value (more on characters a little later)
%f	I want you to print the value of a signed, decimal, floating point number. I want to see the result with a floating decimal point.
%5d	I want you to print the decimal integer in five character positions, right justified.
%6.2f	I want you to print the decimal floating point number in six character positions, right justified and to an accuracy of two decimal places.

The rest of the text is quite straightforward, until we get to the **** character, which marks a formatting instruction. These instructions allow you to tell **printf** to do things like take a new line, move the cursor to the start of the line and etc. The **** is followed by a single character which identifies the required function. In the case of our program we want to take a new line at that point, and that is what n does. Note that unlike some languages, print statements do not automatically take a new line. This means that the programmer has to do more, but does make for greater flexibility.

area) ;

The next piece of information that **printf** needs is the name of the variable to be printed. Note that in this case we do not need to specify the address of the variable, we are just sending its value into the function so that it can be printed out.

printf ("The length of wood needed is : %f feet.\n", wood_length) ;

This line is very similar to the one above it, except that the message and the variable printed are different.

}

This closing brace marks the end of the program. The compiler will now expect to see the end of the file or the start of another function.. If it does not, it means that you have got an unmatched brace somewhere in the program, which will almost certainly mean your program will not work properly. Not all compilers notice this!

Punctuation

That marks the end of our program. One of the things that you will have noticed is that there is an awful lot of punctuation in there. This is vital and must be supplied exactly as C wants it, otherwise you will get what is called a compilation error. This simply indicates that the compiler is too stupid to make sense of what you have given it!

You will quickly get used to hunting for and spotting compilation errors, one of the things you will find is that the compiler does not always detect the error where it takes place, consider the effect of missing out a "(" character. Note that just because the compiler reckons your program is OK is no guarantee of it doing what you want!

Another thing to remember is that the layout of the program does not bother the compiler, the following is just as valid

```
#include <stdio.h>
void main (){ float height, width, area, wood_length ;
scanf ( "%f",
&height ) ; scanf ( "%f", &width ) ; area = 2 * height * width ;
 wood_length = 2 * ( height + width ) * 3.25 ; printf (
"The area of glass needed is : %f metr es.\n"
, area ) ; printf (
"The length of wood needed is : %f feet. \n"
, wood_length ) ;}
```

- although if anyone writes a program which is laid out this way they will get a smart rap on the knuckles from me!

The grammar of programs is something you will pick up as we look at more and more of them....

Variables

Variables and Data

Programs operate on data. A programming language must give you a way of storing the data you are processing, otherwise it is useless. What the data actually means is something that you as programmer decide (see above digression on data).

Types of Variables

Basically there are two types of data:

? Nice chunky individual values, for example the number of sheep in a field, teeth on a cog, apples in a basket.

? Nasty real world type things, for example the current temperature, the length of a piece of string, the speed of a car.

In the first case we can hold the value exactly; you always have an exact number of these items, they are *integral*.

When you are writing a specification you should worry about the precision to which values are to be held. Too much accuracy may slow the machine down - too little may result in the wrong values being used.

In the second case we can never hold what we are looking at exactly. Even if you measure a piece of string to 100 decimal places it is still not going to give you its exact length - you could always get the value more accurately. These are *real*. A computer is digital, i.e. it operates entirely on patterns of bits which can be regarded as numbers. Because we know that it works in terms of ons and offs it has problems holding real values. To handle real values the computer actually stores them to a limited accuracy, which we hope is adequate (and usually is).

This means that when we want to store something we have to tell the computer whether it is an integer or a real. In fact it is useful to have several types of data.

Declaration

Robs' Arbitrary Standard For Coloured Boxes!

You tell C about something you want to store by declaring it. The declaration also identifies the type of the thing we want to store. Think of this as C creating a box of a particular size, specifically designed to hold items of the given type. We identify what type of data a particular box can hold by making it a certain colour, according to RASFCB integer boxes are coloured red and real (or floating point) boxes are coloured green.

Note that C is often called weakly typed. This does not mean that you do not press the keys very hard (ho ho), what it means is that C does not fuss about what you put in each box. If you have a red box (of type integer), and then ask C to put a green value into it (of type real or floating point), C will not complain that what you want to do is meaningless, it will just do it. Other languages, for example PASCAL, get all hot under the collar when you try to do this, they are called strongly typed. Weakly typed languages let you do exactly what you want so, provided you know what you are doing, everything will work out OK.

We will consider just three types for the moment, there are others available but you can get by with these for now:

int variables

Note that we can go one further negative than positive. This is because the numbers are stored using "2's complement" noation.

A type of box which can hold only integer values, i.e. no fractional part. The precise range of integers which are supported varies from one version of C to another, in the version we are using it goes from **-32768** to **+32767**

float variables

A type of box which can hold a real (i.e. floating point) number. These are held to limited precision, again the precision and range you get varies from one version of C to another, in the version of C we are using it goes from **3.4E-38** to **3.4E+38**.

char variables

A type of box which can hold a single character. We have already come across characters when we looked at filenames and the like. A character is what you get when you press a key on a keyboard.

Missing Types

If you are used to other languages you might think that there are some storage types missing, for example there is no way to store strings of characters. In C, unlike BASIC, you have to manage the storage of strings yourself - however there are a large number of built in facilities to make like easier. We will come to these later.

Another missing type is that used to store logical values, i.e. either **TRUE** or **FALSE**. C does not provide this facility, but uses the convention throughout that **0** means false and any other value means true. There are operators that can be used to provide logical combinations in C, but they will work on any non-real data type.

Variable Declaration

Before you can use a particular box you have to tell the compiler about it. C will not just create boxes for you; it has to know the name of the box and what you are going to put into it. You tell C about a box by declaring it.

When the compiler is given a declaration it says something along the lines of:

Here is the name of something which we want to store. I will create a box of the type requested, paint it the correct colour and write the name on it. Later on I will put things in the box and get things out of it. I will put the box on a shelf out of the way for now.

Such a box is often called a *variable*. All languages support different types of variables, all have types equivalent to the C ones.

Remember that all we have is a box. The box is not empty at the moment, but there is nothing of interest in it; the contents are what we call undefined. If you use the contents of an undefined variable you can look forward to your program doing something different each time you run it!

C paints the name on the box using a stencil. There are only a few stencils available, so the characters that you can use to name a variable are limited. Also, because of the design of the stencils themselves, there are some rules about how the names may be formed, namely:

> ? All variable names must start with a letter.

> ? After the letter you can have either letters or numbers or the underscore "_" character.

The length of the name allowed depends on the version of C you are using. You can use amazingly long names in any version of C, the important thing to remember is that only a certain number of characters are looked at, i.e.

very_long_and_impressive_C_variable_called_fred

and

very_long_amd_impressive_C_variable_called_jim

- would probably be regarded as the same in most versions of C.

Upper and lower case letters are different, i.e. **Fred** and **fred** are different variables.

Here are a few example declarations, one of which are not valid (see if you can guess which one and why) :

```
int fred ;
float jim ;
char 29yesitsme ;
```

One of the golden rules of programming, along with *"always use the keyboard with the keys uppermost"* is:

> *Always give your variables meaningful names.*

According to the Mills and Boon romances that I have read, the best relationships are meaningful ones.

Giving Values to Variables

Once we have got ourselves a variable we now need to know how to put something into it, and get the value out. C does this by means of an assignment. There are two parts to an assignment, the thing you want to assign and the place you want to put it, for example consider the following:

```
void main ( void )
{
    int first, second, third ;
    first = 1 ;
    second = 2 ;
    second = second + first ;
}
```

The first part of the program should be pretty familiar by now. Within the **main** function we have declared three variables, **first**, **second** and **third**. These are each of integer type.

The last three statements are the ones which actually do the work. These are assignment statements. An assignment gives a value to a specified variable, which must be of a sensible type (note that you must be sensible about this because the compiler, as we already know, does not know or care what you are doing). The value which is assigned is an expression. The equals in the middle is there mainly do confuse us, it does not mean equals in the numeric sense, I like to think of it as a gozzinta (see above). Gozzintas take the result on the right hand side of the assignment and drop it into the box on the left, which means that:

2 = second + 1 ;

is a piece of programming naughtiness which would cause all manner of nasty errors to appear.

Expressions

An expression is something which returns a result. We can then use the result as we like in our program. Expressions can be as simple as a single value and as complex as a large calculation. They are made up of two things, *operators* and *operands*.

Operands

Operands are things the operators work on; They are usually constant values or the names of variables. In the program above **first**, **second**, **third** and **2** are all operands.

Operators

Operators are the things which do the work; They specify the operation to be performed on the operands. Most operators work on two operands, one each side. In the program above **+** is the only operator.

Here are a few example expressions:

2 + 3 * 4
-1 + 3
(2 + 3) * 4

These expressions are worked out (evaluated) by C moving from left to right, just as you would yourself. Again, just as in traditional maths all the multiplication and division is performed first in an expression, followed by the addition and subtraction.

C does this by giving each operator a priority. When C works out an expression it looks along it for all the operators with the highest priority and does them first. It then looks for the next ones down and so on until the final result is obtained. Note that this means that the first expression above will therefore return **14** and not **20**.

If you want to force the order in which things are worked out you can put brackets around the things you want done first, as in the final example. You can put brackets inside brackets if you want, provided you make sure that you have as many open ones as close ones. Being a simple soul I tend to make things very clear by putting brackets around everything.

It is probably not worth getting too worked up about this expression evaluation as posh people call it, generally speaking things tend to be worked out how you would expect them.

For completeness here is a list of all operators, what they do and their precedence (priority). I am listing the operators with the highest priority first.

op.	use
-	unary minus, the minus that C finds in negative numbers, e.g. -1. Unary means applying to only one item.
*	multiplication, note the use of the * rather than the more mathematically correct but confusing x.
/	division, because of the difficulty of drawing one number above another on a screen we use this character instead
+	addition, no problems here.
-	subtraction. Note that we use exactly the same character as for unary minus.

This is not a complete list of all the operators available, but it will do for now. Because these operators work on numbers they are often called the numeric operators.

Types of Data in Expressions

When C performs an operator, it makes a guess as to the type of the result that is to be produced. Essentially, if the two operands are integer, it says that the result should be integer, if the two are floating point, it says that the result should be floating point. This can lead to problems, consider the following :

1/2
1/2.0

You might think that these would give the same result. Not so. The compiler thinks that the first expression, which involves only integers, should give an integer result. It therefore would calculate this to be the integer value **0** (the fractional part is always truncated). The second expression, because it involves a floating point value would however be evaluated to give a floating point result, the correct answer of **0.5**

Casting

We can force C to regard a value as being of a certain type by the use of casting. A cast takes the form of an additional instruction to the compiler to force it regard a value in a particular way. You cast a value by putting the type you want to see there in brackets before it. For example :

```
#include <stdio.h>
void main ( void )
{
    int i = 3, j = 2 ; float fraction ;
    fraction = (float) i / (float) j ;
    printf ( "fraction : %f\n", fraction ) ;
}
```

The **(float)** part of the above tells the compiler to regard the values in the integer variables as floating point ones, so that we get **1.5** printed out rather than **1**.

When you cast, which you need to do occasionally, remember that casting does not affect the actual value, just how C regards the number. As we saw above, each type of variable has a particular range of possible values, and the range of floating point values is much greater than that for integers. This means that if you do things like :

```
int i ;
i = (int) 12345678.999 ;
```

I do not think of this as a failing in C. It gives you great flexibility, at the cost of assuming you know what you are doing....

- the cast is doomed to fail. The value which gets placed in **i** will be invalid. Nowhere in C does the language check for mistakes like this. It is up to you when you write your program to make sure that you never exceed the range of the data types you are using - the program will not notice but the user certainly will!

Getting Values into the Program

We have used the **printf** (print-formatted) standard input/output procedure from stdio.h to do the output, what we need is an equivalent procedure to do the input. C has got one of these, it is called **scanf** (scan-formatted). However, before we can turn it loose we have a little problem to solve; Where does **scanf** put the value that it fetches. "That is easy" you say, simply give it the name of the variable and **scanf** will know where to go. Wrong! **scanf** is very stupid. It cannot understand the names of variables. Why should it? All it was created for is to take something from one place (the keyboard) and put it somewhere else (a location). **scanf** does not want to know what the variable is called, all it needs to know is where the variable lives.

When we talk about C variables we mean a box which the compiler creates for us, with a name nicely painted on it. The compiler makes a box like this and then puts it on a shelf somewhere safe. The compiler then knows that when we say "**i**" we really mean the box with **i** written on it. **scanf** is simply a function that we call to fetch a value and stick it somewhere else, another kind of removal man. It needs to be told which box to put the result in. C calls referring to a thing by its location rather than its name as pointing.

Nice children have been brought up knowing that it is rude to point. In C things are rather different, you have to point to be able to do anything useful. (I am sure nanny would understand).

For more about pointers, see the chapter about Functions.

When we want **scanf** to fetch something for us we have to give it a pointer. You can regard a pointer as a tag on the end of a piece of rope. The rope is tied to one of our boxes. All **scanf** has to do is follow the piece of rope to a box and then put the value into that box.

You tell the C compiler to generate a pointer to a particular variable by putting an ampersand **"&"** in front of the variable name, i.e.

x   means the value of the variable **x**

&x   means a pointer to the box where **x** is kept, i.e. the address of **x**.

scanf looks very like **printf**, in that it is a format string followed by a list of items, but in this case the items pointers to variables, rather than values, for example:

    scanf ("%d %d %d", &i, &j, &k) ;

The more adventurous amongst you may be wondering what happens if we leave off the ampersands by mistake. This can lead to one of two things:

1. The compiler noticing that you have been a naughty programmer and taking you behind the bike sheds for a meaningful discussion.

2. **scanf** taking the value which it gets and stuffing it into a spurious chunk of memory, with the consequent and hilarious total failure of program and possibly computer.

A very big chunk of C involves address and pointer juggling. I make no apologies for this, it is one of the things that makes C the wonderful, fun loving, language that it is. However it can also be a little hard on the grey matter. Just keep in mind that only the C compiler can handle sticking values into variables. Everything else does not know the name, and can only talk in terms of following pointers to boxes.

Writing a Program

Comments

When the C compiler sees the "**/***" sequence which means the start of a comment it says:

*Aha! Here is a piece of information for greater minds than mine to ponder. I will ignore everything following until I see a */ which closes the comment.*

Be generous with your comments. They help to make your program much easier to understand. You will be very surprised to find that you quickly forget how you got your program to work. You can also use comments to keep people informed of the particular version of the program, when it was last modified and why, and the name of the programmer who wrote it - if you dare!

Program Flow

The program above is very simple, it runs straight through from the first statement to the last, and then stops. Often you will come across situations where your program must change what it does according to the data which is given to it. Basically there are three types of program flow :

1. straight line
2. chosen depending on a given condition
3. repeated according to a given condition

Every program ever written is composed of the three elements above, and very little else! You can use this to good effect when designing an overall view of how your program is going to work. At the moment we have only considered programs which run in a straight line and then stop. The path which a program follows is sometimes called its "thread of execution". When you call a function the thread of execution is transferred into the function until it is complete.

Conditional Execution - if

The program above is nice, in fact our customer will probably be quite pleased with it. However, it is not perfect. The problem is not with the program, but with the user.

If you give the program a window width of **-1** it goes ahead and works out a stupid result. Our program does not have any checking for invalid widths and heights. The user might have grounds for complaint if the program fails to recognise that he has given a stupid value, in fact a number of cases are currently being fought in the United States courts where a program has failed to recognise invalid data, produced garbage and caused a lot of damage.

What we want to do is notice the really stupid replies and tell the user that he has done something dubious. In our program specification, which we give the customer, we have said something like:

The program will reject window dimensions outside the following ranges:

width less than 0.5 metres
width greater than 5.0 metres
height less than 0.75 metres
height greater than 3.0 metres

This means that we have done all we can; If the program gets **1** rather than **10** for the width then that is the users' problem, the important thing from our point of view is that the above specification stops us from being sued!

In order to allow us to do this the program must notice naughty values and reject them. To do this we need can use the construction:

if (condition)
    statement or block we do if the condition is true
else
    statement or block we do if the condition is false

The condition determines what happens in the program. So what do we mean by a condition? C is very simple minded about this, essentially it says that any condition which is non-zero is true, any condition which is zero is false. A condition is therefore really something which returns a number, for example:

if (1)
    printf ("hello mum") ;

- is valid, although rather useless as the condition is always true, so "**hello mum**" is always printed (note that we left the else portion off - this is OK because it is optional).

Conditions and Relational Operators

To make conditions work for us we need a set of additional relational operators which we can use to make choices. Relational operators work on operands, just like numeric ones. However any expression involving them can only produce two values, **0** or **1**. Zero if the expression is not true, one if it is. Relational operators available are as follows:

==

equals. If the left hand side and the right hand side are equal the expression has the value **1**. If they are not equal the value is **0**.

> **4 == 5**

- would return **0,** i.e. false. Note that it is not meaningful to compare floating point values in this way. Because of the fact that they are held to limited precision you might find that conditions fail when they should not for example the following equation :

> **x = 3.0 * (1.0 / 3.0) ;**

- may well result in x containing **0.99999999**, which would mean that :

> **(x == 1.0)**

If you want to compare floating point values subtract them and see if the difference is very small.

- would be false - even though mathematically the test should return true.

!=

not equal. The reverse of equal. If they are not equal the expression has the value **1**, if they are equal it has the value **0**. Again, this test is not suitable for use with floating point numbers.

<

less than. If the item on the left is less than the one on the right the value of the expression is 1. If the left hand value is larger than or equal to the right hand one the expression gives **0**. It is quite valid to compare floating point numbers in this way.

>

greater than. If the expression on the left is greater than the one on the right the result is **1**. If the expression on the left is less than or equal to the one on the right the result is **0**.

<=

less than or equal to. If the expression on the left is less than or equal to the one on the right you get 1, otherwise you get 0.

>=

greater than or equal to. If the value on the left is greater than or equal to the one on the right you get **1**, otherwise it is **0**.

!

not. This can be used to invert a particular value or expression, for example you can say **!1**, which is **0**, or you could say: **!(x=y)** - which means the same as **(x!=y).** You use not when you want to invert the sense of an expression.

Combining Logical Operators

Sometimes we want to combine logical expressions, to make more complicated choices, for example to test for a window width being valid we have to test that it is greater than the minimum and less than the maximum. C provides additional operators to combine logical values :

&&

and. If the expressions each side of the **&&** are true the result of the **&&** is true. If one of them is false the result is false, for example

    (width > 0.5) && (width < 5.0))

- this would be true if the width was valid according to our above description.

||

or. If either of the expressions each side of the **||** are true the result of the whole expression is true. The expression is only false if both expressions are false, for example :

    (width <= 0.5) | (width >= 5.0))

De Morgans theorem is the basis of this.

- this would be true if the width was invalid according to our above description. Note that if we put an or in we have to also flip the conditional operators around as well.

Using these operators in conjunction with the **if** statement we can make decisions and change what our program will do in response to the data we get.

Lumping Code Together

We have decided that if the user gives a value outside our allowed range an error is generated and the value is then set to the appropriate maximum or minimum. To do this we have to do two statements which are selected on a particular condition, one to print out the message and the other to perform an assignment. You can do this by using the **{** and **}** characters. A number of statements lumped together between **{** and **}** characters is regarded as a single statement, so we do the following:

```
if ( width > 5.0 )      {
    printf ( "Width too big, using maximum \n" ) ;
    width = 5.0 ;
}
```

The two statements are now a block, which is performed only if width is greater than **5.0**. You can lump many hundreds of statements together in this way, the compiler does not mind. You can also put such blocks of code inside other blocks, this is called nesting.

The number of **{** and **}** characters must agree in your program, otherwise you will get strange and incomprehensible errors when the compiler hits the end of the file in the middle of a block or reaches the end of your program half way down your file!

I make things much easier to understand by indenting a particular block by a number of spaces, i.e. each time I open a block with the **{** character I move my left margin in a little way. I can then see at a glance whereabouts I am in the levels at any time.

Magic Numbers and #define

A magic number is a value with a special meaning. It will never be changed within the program, it is instead a constant which is used within it. When I write my glazing program I will include some magic numbers which give the maximum and minimum values for heights and widths. I could just use the values **0.5**, **5.0**, **0.75** and **3.0** - but these are not packed with meaning and make the program hard to change. If, for some reason, my maximum glass size becomes **4.5** metres I have to look all through the program and change only the appropriate values. I do not like the idea of "magic numbers" in programs, what I would like to do is replace each number with something a bit more meaningful.

We can do this by using a part of the compiler called the C pre-processor. The pre-processor sits between your program and the compiler. It can act as a kind of filter, responding to directives in your program, and doing things to the text before the compiler sees it. We are just going to look at the one directive at the moment, **#define**:

    #define PI 3.141592654

Whenever the pre-processor sees **PI** it sends the compiler **3.141592654**. This means that you can do things like:

    circ = rad * 2 * PI ;

The item which follows the **#define** directive should be a sequence of characters. You then have a space, followed by another sequence of characters. Neither of the two sequences are allowed to contains spaces, this would get the pre-processor confused. Anywhere you use a magic number you should use a definition of this form, for example:

    #define MAX_WIDTH 5.0

This makes your programs much easier to read, and also much easier to change.

There is a C convention that you always give the symbol you are defining in CAPITAL LETTERS. This is so that when you read your program you can tell which things have been defined.

Note that the **#define** directive is not intelligent, and you can therefore stop your program from working by getting your definition wrong. Consider the effect of:

    #define MAX_WIDTH *this*will*cause*an*explosion!

There are loads more pre-processor directives which you can use, the other one which we have already seen is **#include**.

We can therefore modify our double glazing program as follows:

```
/* Double Glazing 2 */
/* This program calculates glass area and wood */
/* required by a double glazing salesman. */
/* Version 22.15 revision level 1.23 */
/* Rob Miles - University of Hull - 13/11/89 */

#include <stdio.h>
```

```
#define MAX_WIDTH 5.0
#define MIN_WIDTH 0.5
#define MAX_HEIGHT 3.0
#define MIN_HEIGHT 0.75

void main ()
{
        float width, height, glassarea, woodlength ;

        printf ( "Give the width of the window  : " ) ;
         scanf ( "%f", &width ) ;
        if (width < MIN_WIDTH) {
                printf ( "Width is too small. \n\n" ) ;
                width = MIN_WIDTH ;
        }
        if (width > MAX_WIDTH) {
                printf ( "Width is too large. \n\n" ) ;
                width = MAX_WIDTH ;
        }
         scanf ( "%f", &height ) ;
        if (height < MIN_HEIGHT) {
                printf ( "Height is too small. \n\n" ) ;
                height = MIN_HEIGHT ;
        }
        if (height > MAX_HEIGHT) {
                printf ( "Height is too large. \n\n" ) ;
                height = MAX_HEIGHT ;
        }

        woodlength = 2 * (width + height) ;
        glassarea = width * height  ;
        printf ( "Glass : %f Wood : %f\n\n", woodlength, glassarea ) ;
}
```

This program fulfills our requirements. It will not use values incompatible with our specification. However I would still not call it perfect. If our salesman gives a bad height the program stops and needs to be re-run, with the height having to be entered again.

What we would really like is a way that we can repeatedly fetch values for the width and height until we get one which fits.

C allows us to do this by providing a looping constructions.

Loops

Conditional statements allow you to do something if a given condition is true. However often you want to repeat something while a particular condition is true, or a given number of times.

C has three ways of doing this, depending on precisely what you are trying to do. Note that we get three methods not because we need three but because they make life easier when you write the program (a bit like an attachment to our chainsaw to allow it to perform a particular task more easily). Most of the skill of programming involves picking the right tool or attachment to do the job in hand. (the rest is finding out why the tool didn't do what you expected it to!).

In the case of our program we want to repeatedly get numbers in until while we are getting duff ones, i.e. giving a proper number should cause our loop to stop.

This means that if we get the number correctly first time the loop will execute just once. You might think that I have pulled a fast one here, all I have done is change:

Get values until you see one which is OK

into

Get values while they are not OK

Part of the art of programming is changing the way that you think about the problem to suit the way that the programming language can be told to solve it. Further details can be found in Zen and the Art of 68000 Assembler from Psychic Press at £150.

do -- while loop

In the case of our little C program we use the do -- while construction which looks like this:

```
do
    statement or block
while (condition) ;
```

This allows us to repeat a chunk of code until the condition at the end is true. Note that the test is performed after the statement or block, i.e. even if the test is bound to fail the statement is performed once.

A condition in this context is exactly the same as the condition in an if statement, raising the intriguing possibility of programs like:

```
#include <stdio.h>

void main ()
{
    do
            printf ( "hello mum \n" ) ;
    while (1) ;
}
```

This is a perfectly legal C program. How long it will run for is an interesting question, the answer contains elements of human psychology, energy futures and cosmology, i.e. it will run until:

1. You get bored with it.
2. Your electricity runs out.
3. The universe implodes.

This is a chainsaw situation, not a powerful chainsaw situation. Just as it is possible with any old chainsaw to cut off your leg if you try really hard so it is possible to use any programming language to write a program which will never stop. It reminds me of my favourite shampoo instructions:

1. Wet Your Hair
2. Add Shampoo and Rub vigorously.
3. Rinse with warm water.
4. Repeat.

I wonder how many people there are out there still washing their hair at the moment.

while loop

Sometimes you want to decide whether or not to repeat the loop before you perform it. If you think about how the loop above works the test is done after the code to be repeated has been performed once. For our program this is exactly what we want, we need to ask for a value before we can decide whether or not it is valid. In order to be as flexible as possible C gives us another form of the loop construction which allows us to do the test first:

while (condition)
statement or block

Note that C makes an attempt to reduce the number of keys you need to press to run the program by leaving out the word **do**. (if you put the **do** in the compiler will take great delight in giving you an error message - but you had already guessed that of course!).

for loop

Often you will want to repeat something a given number of times. The loop constructions we have given can be used to do this quite easily:

```
#include <stdio.h>
void main ()
{
    int i ;
    i = 1 ;
    while ( i < 11) {
            printf ( "hello\n" ) ;
            i = i + 1 ;
    }
}
```

The variable which controls things is often called the control variable, and is usually given the name i.

This useless program prints out **hello** 10 times. It does this by using a variable to control the loop. The variable is given an initial value (**1**) and then tested each time we go around the loop. The control variable is then increased for each pass through the statements. Eventually it will reach 11, at which point the loop terminates and our program stops.

C provides a construction to allow you to set up a loop of this form all in one:

```
for ( setup ; finish test ; update ) {
    things we want to do a given
    number of times
}
```

We could use this to re-write the above program as:

```
#include <stdio.h>
void main ()
{
    int i ;
    for ( i=0 ; i != 11 ; i = i+1 ) {
            printf ( "hello\n" ) ;
    }
}
```

The setup puts a value into the control variable which it will start with. The test is a condition which must be true for the for -- loop to continue. The update is the statement which is performed to update the control variable at the end of each loop. Note that the three elements are separated by semicolons. The precise sequence of events is as follows:

1. Put the setup value into the control variable.
2. Test to see if we have finished the loop yet and exit to the statement after the for loop if we have.
3. Perform the statements to be repeated.
4. Perform the update.
5. Repeat from step 2.

Writing a loop in this way is quicker and simpler than using a form of **while** because it keeps all the elements of the loop in one place, instead of leaving them spread about the program. This means that you are less likely forget to do something like give the control variable an initial value, or update it.

If you are so stupid as to mess around with the value of the control variable in the loop you can expect your program to do stupid things, i.e. if you put **i** back to **0** within the loop it will run forever.....

Breaking Out of Loops

Sometimes you may want to escape from a loop whilst you are in the middle of it, i.e. your program may decide that there is no need or point to go on and wishes to tunnel out of the loop and continue the program from the statement after it.

You can do this with the **break** statement. This is a command to leap out of the loop immediately. Your program would usually make some form of decision to quit in this way. I find it most useful so that I can provide a get the hell out of here option in the middle of something, for example in the following program snippet the variable **aborted**, normally **0** becomes **1** when the loop has to be abandoned and the variable **runningOK**, normally **1**, becomes **0** when it is time to finish normally.

```
while (runningOK) {
    complex stuff
    ....
    if (aborted) {
            break ;
    }
    ....
    more complex stuff
    ....
}
....
bit we get to if aborted becomes true
....
```

Note that we are using two variables as switches, they do not hold values as such, they are actually used to represent states within the program as it runs. This is a standard programming trick that you will find very useful.

You can break out of any of the three kinds of loop. In every case the program continues running at the statement after the last statement of the loop.

Going Back to the Top of a Loop

Every now and then you will want to go back to the top of a loop and do it all again. This happens when you have gone as far down the statements as you need to. C provides the **continue** statement which says something along the lines of:

Please do not go any further down this time round the loop. Go back to the top of the loop, do all the updating and stuff and go around if you are supposed to.

In the following program the variable **Done_All_We_Need_This_Time** is set when we have gone as far down the loop as we need to.

```
for ( item = 1 ; item < Total_Items ; item=item+1 ) {
    .....
    item processing stuff
    ....
    if (Done_All_We_Need_This_Time) {
            continue ;
    ....
    additional item processing stuff
    ....
}
```

The **continue** causes the program to re-run the loop with the next value of item if it is OK to do so. You can regard it as a move to step 2 in the list above.

More Complicated Decisions

We can now think about using a loop to test for a valid width or height. Essentially we want to keep asking the user for a value until we get one which is OK; i.e. if you get a value which is larger than the maximum or smaller than the minimum ask for another.

To do this we have to combine two tests to see if the value is OK. Our loop should continue to run if:

width > MAX_WIDTH

or

width < MIN_WIDTH

To perform this test we use one of the logical operators described above to write a condition which will be true if the width is invalid:

((width < MIN_WIDTH) || (width > MAX_WIDTH))

- note the profuse use of brackets. You **must** put these in.

Complete Glazing Program

This is a complete solution to the glazing problem. It uses all the tricks mentioned above, plus a few which are covered below.

```
/* Complete Double Glazing Program */
/* Rob Miles Nov. 1990  */

#include <stdio.h>

/* Define our window size range */
#define MAX_HEIGHT 3.0
#define MAX_WIDTH 5.0
#define MIN_HEIGHT 0.75
#define MIN_WIDTH 0.5

/* Define a few costs */
#define COST_TO_MAKE 2.00
#define WOOD_COST 2.00
```

```c
#define GLASS_COST 3.00
#define MARKUP_FACTOR 2.25

/* Define the maximum number of windows on our house  */
#define MAX_WINDOWS 10

/* Program variables :  */
/* width - width of current window  */
/* height - height of current window  */
/* window_cost - cost to make the window      */
/* window_sell - amount we sell the window for        */
/* house_cost - cost to do the whole house  */
/* house_sell - amount we sell the house job for  */
float width, height, window_cost, window_sell, house_cost, house_sell ;

/* no_of_windows - number of windows in the house            */
/* window_count - counter for current window          */
int no_of_windows, window_count ;

void main ()
{
        printf ( "Double Glazing House Calculator \n" ) ;
        printf ( "Rob Miles November 1990 \n" ) ;

        do {
                printf ( "Give the number of windows : " ) ;
                scanf ( "%d", &no_of_windows ) ;
        } while ( (no_of_windows <= 0) || (no_of_windows > MAX_WINDOWS) ) ;

        house_cost = 0.0 ;
        house_sell = 0.0 ;

        for (window_count = 1 ; window_count <= no_of_windows ; window_count++ ) {
                printf ( "Window %d details.\n", window_count ) ;
                do {
                        printf ( " enter the width : " ) ;
                        scanf ( "%f", &width ) ;
                } while ( (width < MIN_WIDTH) || (width > MAX_WIDTH) ) ;
                do {
                        printf ( " enter the height : " ) ;
                        scanf ( "%f", &height ) ;
                } while ( (height > MIN_HEIGHT) ||  (height < MAX_HEIGHT) ) ;
                window_cost = WOOD_COST * 2 * (width + height) ;
                window_sell = window_cost * MARKUP_FACTOR ;
                printf ( "Window %d costs %.2f, sells for %.2f. \n\n",
                        window_count, window_cost, window_sell ) ;
                house_cost += window_cost ;
                house_sell += window_sell ;
        }
        printf ( "\nTotal cost to do all %d windows : %.2f. \n", no_of_windows, house_cost ) ;
        printf ( "\nTotal sale price for all %d windows : %.2f. \n", no_of_windows, house_sell ) ;
}
```

Operator Shorthand

So far we have looked at operators which appear in expressions and work on
two operands, e.g.

window_count = window_count + 1

In this case the operator is **+** and is operating on the variable **window_count** and the value **1**. The purpose of the above statement is to add **1** to the variable **window_count**. However, it is a rather long winded way of expressing this, both in terms of what we have to type and what the computer will actually do when it runs the program. C allows us to be more terse if we wish, the line :

window_count++

- would do the same thing. We can express ourselves more succinctly and the compiler can generate more efficient code because it now knows that what we are doing is adding one to a particular variable. The **++** is called a unary operator, because it works on just one operand. It causes the value in that operand to be increased by one. There is a corresponding **--** operator which can be used to decrease (decrement) variables. You can see examples of this construction in the for loop definition in the example above.

The other shorthand which we use is when we add a particular value to a variable. We could put :

house_cost = house_cost + window_cost

This is perfectly OK, but again is rather long winded. C has some additional operators which allow us to shorten this to:

house_cost += window_cost

The **+=** operator combines addition and the assignment, so that the value in **house_cost** is increased by **window_cost**. Some other shorthand operators are:

a += b   the value in a is replaced a+b.

a -= b   the value in a is replaced by a - b.

a /= b   the value in a is replaced by a / b.

a *= b   the value in a is replaced by a * b.

There are other combination operators, I will leave you to find them!

Statements and Values

One of the really funky things about C is that all statements return a value, which you can use in another statement if you like. Most of the time you will ignore this value, which is OK, but sometimes it can be very useful, particularly when we get around to deciding things (see later). In order to show how this is done, consider the following:

i = (j=0) ;

This is perfectly legal (and perhaps even sensible) C. It has the effect of setting both **i** and **j** to **0**. An assignment statement always returns the value which is being assigned (i.e. the bit on the right of the gozzinta). This value can then be used as a value or operand. If you do this put brackets around the statement which is working as a value, this makes the whole thing much clearer for both you and the compiler!

When you consider operators like **++** there is possible ambiguity, in that you do not know if you get the value before or after the increment. C provides a way of doing it either way, depending on which effect you want. You determine whether you want to see the value before or after the sum by the position of the

++ :

i++   means give me the value before the increment.

++i   means give me the value after the increment.

As an example :

```
int i = 2, j ;
j = ++i ;
```

- would make **j** equal to **3**. The other special operators, **+=** etc all return the value after the operator has been performed.

One neat place to use this facility is when you are updating a value and testing it at the same time, for example instead of :

```
i = i - 1 ;
if ( i == 0 ) {
    printf ( "Finished\n" ) ;
}
```

you could put :

```
if ( --i == 0 ) {
    printf ( "Finished\n" ) ;
}
```

An important thing to note is that not only does this facility make the program shorter, and therefore help the programmer, but it also allows the compiler to produce much faster programs. In the first instance the compiler would tend to produce a larger and slower program, because it would treat the two statements as separate. In the second case, because it knows that we are going to use the value of the sum in an expression it might not have to fetch it again.

This is why people say that the C language is faster than some others; it allows us to write more expressive programs which can be mapped more easily onto the actual machine which will run them. The bad new is that this expressiveness can also be used to write incomprehensible programs as far as people are concerned. In the final analysis, I am happier if the program is a little less efficient, but is easier for people to understand!

Neater Printing

Note that the way that a number is printed does not affect how it is stored in the program, it just tells printf to cut short the printing at a particular point.

If you have run any of the above programs you will by now have discovered that the way in which numbers are printed leaves much to be desired. Integers seem to come out OK, but real numbers seem to have a particularly large number of decimal places, which are not required for our program. You can give the **printf** function more information about how a value is to be printed, so that the values that come out look a lot better. For any item you can give the number of character positions to be used to print the number. In addition, for floating point values you can specify the precision, i.e. how many decimal places to print.

You tell **printf** what you want by putting values after the **%** but before the letter which identifies the type of variable to be printed. The general form is:

%width.precisiontype

Note that if you are printing integer types you do not need to give the point or the precision value, just the width.

Here are some examples, with explanations:

Format   Function

%5d	print the decimal integer in a space five characters wide. If the value is smaller than five characters, pad out with extra spaces.
%6.2f	print the floating point value in a space six characters wide, with two decimal places.
%.2f	print the floating point value in a space wide enough to it, with two decimal places.

There are other ways in which you can control the printing process, for example you can select whether "pad out" spaces are inserted to the left or right of the value. I will leave you to find out these! Note also that you can specify the print width of any item, even a piece of text, which makes printing in columns very easy.

Functions

Functions So Far

We have already come across the functions **main**, **printf** and **scanf**. When you write a program of any size you will find that you can use functions to make life a whole lot easier.

In the glazing program above we spend a lot of time checking the values of inputs and making sure that they are in certain ranges. We have exactly the same piece of code to check widths and heights. If we added a third thing to read, for example frame thickness, we would have to copy the code a third time. This is not very efficient, it makes the program bigger and harder to write. What we would like to do is write the checking code once and then use it at each point in the program. To do this you need to define a function to do the work for you.

Functions are identified by name and can return a single value. They act on parameters, which are used to pass information into them. In our case we want a function which is given the maximum and minimum values which something can have, and returns a value in that range.

We need to tell the compiler the name of our function, what parameters it has, and the type of information it returns.

Function Heading

We do all this in the function heading

float get_ranged_value (float min, float max)

This tells the compiler that we are defining a function called **get_ranged_value**. The first word, **float**, tells the compiler we are defining a

function which will return the value of a floating point number. The second item is the function name, i.e. a name that we wish our function to be known by. You create a function name using exactly the same rules as for a variable name.

The next item is a list of parameter names, enclosed in brackets. We have two parameters, **min** and **max**. We must then tell the compiler the type of these two parameters, in this case they are both floating point numbers.

Note that C does not do that much checking to make sure that you are giving the right kinds of parameters in the correct order. If you get this wrong the program will fail, probably in a very confusing way.

Function Body

We then follow this definition with the body of the function, which is in fact a block, making our complete function as follows:

```
float get_ranged_value ( float  min, float max )
{
    float temp ;
    do {
            printf ( "Give the value : " ) ;
            scanf ( "%f", &temp ) ;
    } while ( (temp < min) || (temp > max) ) ;
    return temp ;
}
```

return

The body of the function looks very like the code which we originally wrote, however there is one additional line :

```
return temp ;
```

This is the means by which the function returns the value to whatever called it. If our function has a type other than void it must return a value of the appropriate type. You return a value by placing it in the program, after the **return** keyword.

You can return at any point during a function. This can be used as a very neat "get the hell out of here".

Calling a Function

When the compiler sees **get_ranged_value** it knows that this refers to a previously defined procedure. It takes the values **MAX_WIDTH** and **MIN_WIDTH** and feeds them into the function. You can use max and min as variables within the function, initially they are set to the values of the appropriate parameter. Any changes to these parameters which you make inside the function are not passed on to the outside world. When you reach the return line in the function this causes the function to finish and pass back the value following the return. Of course you must return something of the correct type, otherwise the compiler will moan at you and nasty things might happen.

In our main program we would write something like :

```
width = get_ranged_value ( MIN_WIDTH, MAX_WIDTH ) ;
```

Local function variables

Note that inside our function we have declared a variable, **temp**. This is used to hold the value which the user types in, so that we can compare it with the maximum and minimum. This is a local variable. We only want to use **temp** during this function, so we tell the compiler that it is only declared inside the function. When **get_ranged_value** finishes the variable is discarded. This is useful for two reasons:

It saves space. We are not reserving space for a variable when we are not going to use it. **temp** only comes into existence when required and the memory it uses is available at other times.

It reduces confusion. Because **temp** is only declared within **get_ranged_value** that is the only place you can use it.

If I mention **temp** in another part of the program the compiler will say that it cannot see it. This means that if someone else has written a function which uses a local variable called **temp** there will be no confusion.

Scope

When talking about variables in this way you often hear the word scope. Any variable has a given *scope*. You could describe the scope of a variable as that portion of a program within which it is meaningful to use that variable.

If you declare a variable outside a function it is effectively accessible everywhere. Such variables are often called global, i.e. their scope is the entire program. You have to be careful when you make a variable global. The fact that it can be used in any part of the program means that it can be corrupted in any part of the program.

Up until now all our variables have been declared to be global. This is not good practice. Only certain values are that important should be global. One part of program design is deciding which of the variables need to be made global. This is particularly valuable when several people are working on one project. If you ever get involved with the writing of a large system the trick is to get together first and decide what global variables to use, then decide on the overall structure, what each section is going to do and how they are going to exchange data. You can then go ahead and start writing your part of the system, secure in the knowledge that you will not be causing anyone else problems.

As an example of variables and scope consider the following:

```
int fred ;                        |
void main ()                      |
{                                 |
    float fred ;        |
    ....                | scope of
    ....                | local fred
}                       |
                                  |
void road ()                      | scope of
{                                 | global
    int jim ;           |         | fred
    ....                | scope of |
    ....                | jim      |
}                       |         |
```

The global **fred** cannot be accessed by the function **main**, any reference to **fred** refers to the local variable. Within **main** this variable is said to be scoped out.

I adopt a little convention when choosing variable names. If the variable is going to be global I start the name with a capital letter, local variables are all lower case, for example:

Window_Total

- would be a global variable whereas

counter

-would be local.

Variables Local to Blocks

You can extend the idea of little local variables even further, in that any block can contain a variable definition which will last for the duration of that block. When that block finishes the variable disappears. This is especially useful if you need a little counter for a job in the middle of a program:

This is a screen clear routine which will work on any machine!

```
{
    int i ;
    for ( i=0 ; i < 25 ; i++ ) {
            printf ( "\n" ) ;
    }
}
```

The variable **i** only exists for the duration of the block, so this loop does not interfere with any other variables called **i** which might be lying around your program. If you have a sudden local need for a variable for a specific task it makes very good sense to create one there and then to do the job. You could argue that it is not very efficient to do this, because the variable is created each time the block is entered, but it makes the program very much clearer and reduces the chances of variable names clashing. Note that you must declare any block variables at the very start of the block.

Full Functions Example

This is the definitive double glazing example:

```
/* Functions Double Glazing Program */
/* Rob Miles Nov. 1990  */

#include <stdio.h>

/* Define our window size range */
#define MAX_HEIGHT 3.0
#define MAX_WIDTH 5.0
#define MIN_HEIGHT 0.75
 #define MIN_WIDTH 0.5

/* Define a few costs */
#define COST_TO_MAKE 2.00
#define WOOD_COST 2.00
#define GLASS_COST 3.00
#define MARKUP_FACTOR 2.25

/* Define the maximum number of windows on our house */
#define MAX_WINDOWS 10

/* Function to get a value and validate its range */
```

```c
/* returns a floating point value within min and max */
float get_ranged_value ( float min, float max )
{
/* get_ranged_value variables : /
/* temp holds the value we are looking at. */
        float temp ;
        do {
                printf ( " enter the value : " ) ;
                scanf ( "%f", &temp ) ;
        } while ( (temp < min) || (temp > max) ) ;
        return temp ;
}

void main ()
{
/* main variables : */
/* width - width of current window */
/* height - height of current window */
/* window_cost - cost to make the window */
/* window_sell - amount we sell the window for */
/* house_cost - cost to do the whole house */
/* house_sell - amount we sell the house job for */

        float width, height, window_cost, window_sell, house_cost, house_sell ;

/* no_of_windows - number of windows in the house */
/* window_count - counter for current window */

        int no_of_windows, window_count ;

        printf ( "Double Glazing House Calculator \n" ) ;
        printf ( "Rob Miles February 1996\n" ) ;

        do {
                printf (  "Give the number of windows in the house : " ) ;
                scanf ( "%d", &no_of_windows ) ;
        } while ( (no_of_windows <= 0) ||
                (no_of_windows > MAX_WINDOWS) ) ;

        house_cost = 0.0 ;
        house_sell = 0.0 ;

        for (     window_count = 1 ;
                 window_count <= no_of_windows ;
                window_count++ ) {

                printf ( "Enter the details of window %d \n", window_count ) ;
                printf ( " enter the width \n" ) ;
                width = get_ranged_value ( MIN_WIDTH, MAX_WIDTH ) ;
                printf ( " enter the heigh t\n" ) ;
                height = get_ranged_value ( MIN_HEIGHT, MAX_HEIGHT ) ;
                window_cost = WOOD_COST * 2 * (width + height) ;
                window_sell = window_cost * MARKUP_FACTOR ;
                printf ( "Window %d costs %.2f to make and sell for %.2f. \n\n",
                        window_count, window_cost, window_sell ) ;
                house_cost += window_cost ;
                house_sell += window_sell ;
        }
```

```
        printf ( "\nTotal cost for %d windows : %.2f.\n",
                no_of_windows, house_cost ) ;
        printf ( "\nTotal sale price for %d windows : %.2f.\n",
                no_of_windows, house_sell ) ;
}
```

Pointers

For now we have only looked briefly at pointers, just using them to tell **scanf** where to put values it fetches for us. However, pointers are much more than this, in fact they are an integral part of C. Unfortunately pointers are also painfully difficult to understand, so do not feel upset if they do not make sense immediately. Just keep looking at the examples until something makes sense!

A pointer is in fact just a different kind of variable. You can regard a normal variable as a box with a name on it. You can regard a pointer as a tag on the end of a piece of rope. By following the rope you can get to a box, into which you can put something

Note that to keep things hunky-dory , C says that pointers can only point to items of a particular type. Think of it like this, all the integer boxes are red, all the float boxes are green and so on. The rope of a pointer is also colour co-ordinated, which means that if you try to tie a piece of red string onto a green box (use an int pointer to point at a float value) the compiler will raise aesthetic objections and complain about your colour scheme!

You tell C that you are declaring a pointer to a type, rather than a variable of that type, by putting a ***** in front of the name of the variable, i.e.

int i;   i   integer variable

**int * ptr;**   ptr   0   pointer to an integer
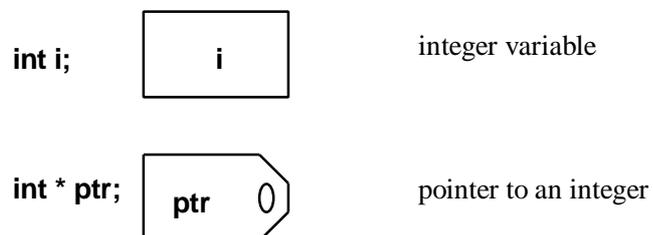
i is defined as an integer variable. **ptr** is defined as a pointer to integers. Remember that neither variable has anything useful in it just after declaration, i.e. **i** contains a silly value and **ptr** points nowhere useful.

To find the address of something, i.e. the value you put in a pointer to make it point at it, we use the **&** character.

We have already met **&**. It means create a a piece of rope which is tied to the box which holds this variable. The following line of code :

ptr = &i ;

- does the following :

int i;   i   integer variable

int * ptr;   ptr   pointer to an integer
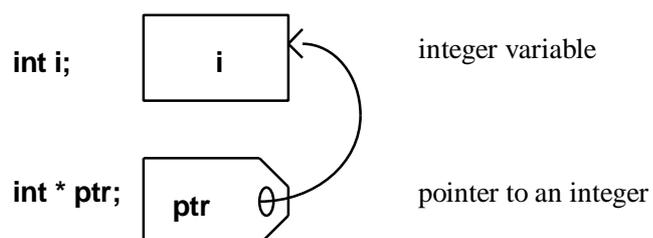
ptr now points at **i**. If I want to change the contents of the box that **ptr** points at I can use the ***** operator to *de-reference* the pointer and get at the contents.

*is new. It means follow the rope on this pointer to the box it is fixed to and use the value in the box.

* and **&** are complementary. You will use **&** when you have a variable which you want to make a pointer to and ***** when you want to refer to the contents of a box which a pointer is pointing at (if this confuses you at the moment just think in terms of pointers being bits of rope tied to boxes and variables being boxes with names on...).

The upshot of all this is that, once we have made **ptr** point to **i**, the following two lines of code have exactly the same effect :

```
i = 99 ;
*ptr = 99 ;
```

If you want to have a quick way of remembering what is going on consider this :

```
int i ;
*(&i) = 99 ;
```

- this is actually legal! It means make a pointer to **i** (that is what the **&** does) and then de-references it (that is what the ***** does). Then put **99** into the place that the pointer to **i** points to, i.e. it is functionally equivalent to :

```
i = 99 ;
```

- we put the brackets in to keep the compiler happy and tell it the order in which things are happening.

As you might expect, if you have not given a value to a pointer then it is tied on to most any old thing, and referring to the contents of an undefined pointers is one of the shorter ways to explode a program....

Consider the following program:

```
#include <stdio.h>
void main ()
{
    int fred, jim ;
    int *pointer1, *pointer2 ;
    pointer1 = &fred ;
    pointer2 = &jim ;
    pointer1 = 99 ; fred = 100 ;
    *pointer2 = *pointer1 ;
    printf ( "The value of fred is %i. \n", fred ) ;
    printf ( "The value of jim is %i. \n", jim ) ;
}
```

Note that the ***** operator can be used on both sides of an assignment, i.e. the contents of the box on the end of this rope will work equally well for putting things into the box as for taking them out. Note also that we are *overloading* the ***** operator. This means that we are using it to mean more than one thing, because we also use ***** to mean multiplication. The C compiler is usually able to make sense of what we give it, and work out what we really mean. However if you are performing multiplication with the contents of pointers it is a good idea to put things inside brackets so as to force the point, e.g.

```
result = (*value_1_pointer) * (*value_2_pointer) ;
```

NULL Pointers

It is often very useful to be able to denote the fact that a pointer does not point to anything useful. For example, if your function is supposed to set a pointer which points to the item it has found, but it does not find the item, you would find it useful to be able to say this your program. In the C standard headers, for example **stdio.h**, there is an address called **NULL**. This address does **not** exist, trying to use it will result in your program doing strange things, but it can be used to mean "this pointer does not point anywhere". The C libraries use this extensively, for example if you try to get hold of a block of memory (see **malloc** later in the notes) but the memory is not available you will instead be given a pointer set to **NULL**, to indicate that the memory could not be found.

You should never put things where **NULL** points, instead you should test the value of the pointer against **NULL:**

```
if ( memory_base == NULL ) {
    printf ( "No more memory!\n\n" ) ;
    exit (1) ;
}
```

Pointers and Functions

You can send values into functions very easily, simply by giving them as parameters, in the case of **get_ranged_value** function we have a parameters of type **float**, into which we can feed values. However, as things stand, a function can only return a single value, via the **return** statement. Suppose we wanted to enhance **get_ranged_value** so that it returned whether or not the user had abandoned the program. We could tell the user that a value of **-1** at any time means abort. However, we now have to return more than one thing from our function, both the result and whether or not the function succeeded.

We can only pass information into functions, and they can only return a single value, so this would seem to be impossible. Fortunately we can use pointers to solve this problem, we give the function a pointer to where we want the result putting, just like we do with **scanf**.

We tell C that a given parameter to a function is a pointer in the same way that we declare a variable which is a pointer. Making these changes leads to the following:

```
/*  If we get this value as input it means give up  */
#define ABORT_VALUE 0

/*  Function to get a value and validate its range  */
/*  returns TRUE if the value was OK and the abort  */
/*  value was not given.  */
/*  puts a floating point value obtained from the  user   */
/*  into the location pointed at by result  */

float get_ranged_value ( float min, float max, float * result ) {
/*  get_ranged_value variables :  */
/*  temp holds the value we are looking at.  */
    float temp ;
    do {
            printf ( " enter the val ue : " ) ;
            scanf ( "%f", &temp ) ;
            if (temp == ABORT_VALUE)
                    return FALSE ;
    } while ( (temp < min) || (temp > max) ) ;
```

```
        *result = temp ;
        return TRUE ;
        }
```

We can call the function and test the result in the following way:

```
if (get_ranged_value (MIN_WIDTH, MAX_WIDTH,
&width)==FALSE )
    break ;
```

Because we can call functions anywhere, even in the middle of comparisons, we can get the value and then immediately test the result of the function. In the code snippit above the program will break out of an enclosing loop if the user gives the abort value for the width.

Once you have got the idea of how pointers can be used to get information in and out of functions it is rather simple. In the meantime just use the above function as an example of how to do it and copy how it works! (there is no dishonour in copying chunks of working program into creations of your own, although for assessment purposes we prefer it if the program is mainly your own work!).

Static Variables

When a program returns from a function C deletes all the local variables. You can think of this as C taking all the boxes off the local shelf and throwing them away.

If the function is ever entered again C will make new boxes, paint their names on them and then put them on the local shelf. This means that a particular local variable will cannot be used to hold a value from one call of a function to the next. If you want a variable to last longer than the length of your function you have to make it global. However, as we have already seen, a very good programming trick is to keep the number of global variables to an absolute minimum.

C gets around this problem by allowing another storage class called **static**. A storage class is an extra piece of information which you give C to tell it more about how you want the variable stored. There are other storage classes, we will come to them later. When you declare a local variable you can put the word static in front of the declaration, for example:

```
static int fred ;
```

When C sees this it makes a box as usual, then it paints the name of the box, along with the name of the function within which it was declared, and then puts this box on a third shelf, the static shelf. Unlike the local shelf, the static one is not emptied when the function finishes. Instead the value is kept in case the function is called again. Note that C has to put the name of the function so that, if several functions have a static variable with the same name, it can find the right one.

With a static variable functions can retain variable values over successive calls.

Arrays

Why We Need Arrays

Your fame as a programmer is now beginning to spread far and wide. The next person to come and see you is the chap in charge of the local cricket team. He would like to you write a program for him which allows the analysis of cricket results. What he wants is quite simple; given a list of cricket scores he wants a list of them in ascending order.

"This is easy" you think. Having agreed the specification and the price you sit down that night and start writing the program. The first thing to do is define how the data is to be stored:

```
int score1, score2, score3, score4, score5, score6, score7
    score8, score9, score10, score11 ;
```

Now you can start putting the data into each variable:

```
printf ( "\nEnter the score for each player in turn. \n\n" ) ;
scanf ( "%i", &score1 ) ;
scanf ( "%i", &score2 ) ;
scanf ( "%i", &score3 ) ;
scanf ( "%i", &score4 ) ;
scanf ( "%i", &score5 ) ;
scanf ( "%i", &score6 ) ;
scanf ( "%i", &score7 ) ;
scanf ( "%i", &score8 ) ;
scanf ( "%i", &score9 ) ;
scanf ( "%i", &score10 ) ;
scanf ( "%i", &score11 ) ;
```

All we have to do next is sort them..... Hmmmm..... This is awful! There seems to be no way of doing it. Just deciding whether **score1** is the largest value would take an if construction with 10 comparisons! Clearly there has to be a better way of doing this, after all, we know that computers are very good at sorting this kind of thing.

C provides us with a thing called an *array*. An array allows us to declare a whole row of a particular kind of box. We can then use things called *subscripts* to indicate which box in the row that we want to use. Consider the following:

```
void main ()
{
    int scores [11] ;
    int i ;
    for (i=0; i<11; i=i+1) {
            scanf ( "%i", &scores [i] ) ;
    }
}
```

The **int scores [11]** tells the compiler that we want to create an array. The bit which defines the size of the array is the **[11]**. When C sees this it says "aha! What we have here is an array". It then gets some pieces of wood and makes a long thin box with 11 compartments in it, each large enough to hold a single integer. It then paints the whole box red - because boxes which can hold integers are red - and then writes "scores" on the side.

Each compartment in the box is called an *element*. In the program you identify which element you mean by putting it's number in square brackets **[ ]** after the array name. This part is called the *subscript*. Note that the thing which makes arrays so wonderful is the fact that you can specify an element by using a variable, as well as a constant. In fact you can use any expression which returns an integer result as a subscript, i.e.

scores [i+1]

- is quite OK.

C numbers the boxes starting at **0**. This means that you specify the first element of the array by giving the subscript **0**. There is consequently no element **scores [11]**. If you look at the part of the program which reads the values into the array you will see that we only count from **0** to **10**. This is very important. An attempt to go outside the array bounds of scores will not cause an error, but could lead to unpredictable things happening!

The real power of arrays comes from our being able to use a variable to specify the required element. By running the variable through a range of values we can then scan through an array with a very small program; indeed to change the program to read in **1000** scores we only have to make a couple of changes:

```
void main ()
{
    int scores [1000] ;
    int i ;
    for (i=0; i<1000; i=i+1) {
            scanf ( "%i", &scores [i] ) ;
    }
}
```

The variable **i** now goes from **0** to **999**, vastly increasing the amount of data we are storing. Note that if I ever create arrays like this I would use **#define** to set the limits on the sizes, so that I can easily change them later.

Sorting

Now we can think about sorting our list of cricket scores. Because of the human passion for order computers spend a lot of their time sorting things. There are vast and dusty tomes in any programming archive on the subject of sorting, and if you look carefully you will find that C actually has a sort routine hidden in one of its libraries. However, we are not going to use a piece of ready written code, we are going to do it all our way - perhaps humming a song made famous by Frank Sinatra.

If you were given 11 numbers to put in order you would look through them for the biggest, write it down and then put down the second largest (which you spotted while you were looking for the biggest) then add the third, which you might have to look for, and then add the others which you can easily pick out from the remainder. In short you would apply massive amounts of processor power (i.e. your brain) in a fairly random, inefficient way. The problem is that when you try to write a program to sort a list of numbers you will find it difficult to describe to yourself how you would do the task, so explaining the task in C is rather difficult!

In the fullness of time you will get used to all the little tricks used when programming, and come to terms with the way that computers "think". Eventually you will be able to think down to the level of the computer, and at that point you can call yourself a programmer!

We are going to use a neat little sorting method called the Bubble Sort. This is a very simple sorting routine which is easy to write and make work. It involves the writing a program which works in following way:

1. Start at the top of the array.

2. Look at the first and the second elements.

3. If they are the wrong way round, swap them.

4. Look at the second and third elements.

5. If they are the wrong way around, swap them.

6. Keep on doing this until you get to the end of the array.

7. Go back to the first step.

The little "program" above does not have a way of stopping, however it is quite easy to decide when you have finished, if you go through the entire list without making any swaps the list must be in the correct order. This solution, whilst a bit boring for a human brain, is the kind of thing that computers just love to bits, a C version of this is:

```c
#include <stdio.h>
#define teamsize 11
void main ()
{
    int scores [teamsize] ; int i, swaps, temp ;
    printf ( "Cricket score sorter 1.0\n" );
    printf ( "Rob Miles\n\n" ) ;
    printf ( "Give each score and press return \n\n" ) ;

    for (i=0; i<teamsize; i++) {
            printf ( "Score for batsman %d : ", i+1 ) ;
            scanf ( "%d", &scores [i] ) ;
    }

    printf ( "\nNow doing the sorting..." ) ;

    do {
            swaps = 0 ; /* clear our swap marker */
            for (i=0 ; i<10; i++) {
                    if ( scores [i] > scores [i+1] ) {
                            /* if we get here we have to swap */
                            temp = scores [i+1] ;
                            scores [i+1] = scores [i] ;
                            scores [i] = temp ;
                            swaps = 1 ; /* mark a swap */
                    }
            }
    } while (swaps != 0) ;

    printf ( "\nThe results are : \n\n" ) ;

    for ( i=0; i<teamsize; i++) {
            printf ( " %d\n", scores [i] ) ;
    }
}
```

The program above actually works! It splits naturally into three chunks, reading
the data, sorting the data and printing the results out. Note the use of a
temporary variable when we swap the elements around. Note also that, because
we compare a value with the one after it in the array, we only count as far as **10**
in the loop which does a pass through the data. If we said **11** the program
would still run, but we might find a strange value from beyond the end of the
array appearing in our data!

Array Types and Sizes

You can have arrays of any type you like, including ones that you create (see
later). There is usually a limit on the maximum length of array that you can
create, but this is usually very large and depends on the version of C that you
are using.

The C compiler needs to know how big an array is going to be, i.e. when you
write your program you must decide the amount of data you want to store. This
can lead to problems, sometimes you do not know in advance how big an array
needs to be. You run the risk of either asking the compiler for too much in
which case your program will use more memory than it needs, or not asking for
enough, in which case your program might fail for lack of room. As we shall

see later, this problem is not insurmountable, but for now we must assume that our program must declare an array of appropriate size, and we must decide that size when we write the program itself.

More Than One Dimension

The array provides us with a way of grabbing a row of boxes, and accessing each box with a subscript. This is fine for linear lists of data, but sometimes you want to hold data of a different shape, perhaps a table or matrix.

You would hit this problem if your cricket playing friend comes back the following week, very pleased with your program and, as users always are, anxious to have it do something else. In this case he wants you to store the results of a whole team season, and then do various statistical things with them which are so beloved of cricket players, in this case find the highest score of the season and player who scored the greatest number of runs in the season.

Resisting the temptation to point him at a spreadsheet program, which is what he really needs, you agree to improve your software. You now have to store several rows of data. If you were writing the results down on paper you might have something like:

Player	Games			
	0	1	2	3
0	10	23	25	80
1	12	34	23	54
2	23	54	6	7
3	16	54	62	8
4	40	23	65	4
5	21	76	8	0
6	4	43	32	2
7	20	12	23	1
8	12	43	22	11
9	3	12	23	15
10	32	2	32	43

From this table you can see that the runs scored in game number 2 by player number 1 are 23 . You identify a particular score by looking along the top for the game number and down the side for the player number. Note that in my table I am numbering the rows and columns starting at 0, i.e. I have a game number 0 and a player number 0, which you might not have in real life (you would need to adjust the values when you display the answers (or make all the dimensions 1 bigger and then waste element 0).

If you think about the data above you can see that it is just a number of one dimensional arrays, i.e. the scores for a number of games which have been put side by side. C lets you create arrays of arrays:

```
#define TEAMSIZE 11
#define SEASONLENGTH 4
....
int scores [SEASONLENGTH]  [TEAMSIZE] ;
```

-what we have here is a number of arrays of length teamsize, one for each match in the season. I can refer to each individual element by giving two subscripts:

scores [2] [1]

- would refer to the score in game 2 of player 1. You can have as many arrays of arrays as you like, the only limit being your brainpower when it comes to sorting out what things mean...

The program to solve our problem would go as follows:

```
/* Cricket team score analyser  */
/* Rob Miles - 1993   */
/* Reads in the scores from 6 cricket seasons and then */
/* prints out some statistics */

#include <stdio.h>

#define TEAMLENGTH 11
#define SEASONLENGTH  7
void main ()
{
        /* define our main storage array */
        int scores [SEASONLENGTH] [TEAMLENGTH] ;

        /* define our totals array */
        int totals [TEAMLENGTH] ;

        /* define our counters   */
        int game_count, player_count ;

        /* topscore holds the highest score so far */
        int topscore ;

        /* first read our data */
        for (    game_count = 0 ;
                game_count < SEASONLENGTH ;
                game_count++) {
                for (    player_count = 0 ;
                        player_count < TEAMLENGTH ;
                        player_count++) {

                        printf ( "Enter score player %d game %d : ",
                                game_count+1, player_count+1 ) ;
                        scanf ( "%d",
                                &scores [game_count][player_count] ) ;
                        }
                }

        /* now zero our totals array */
        for (    player_count = 0 ;
                player_count < TEAMLENGTH ;
                player_count++) {
                totals [player_count] = 0 ;
        }

        /* Now set our initial to p score */
        topscore = 0 ;
```

```
/* Now work out our results */
for (    game_count = 0 ;
         game_count < SEASONLENGTH ;
         game_count++) {
         for (    player_count = 0 ;
                  player_count < TEAMLENGTH ;
                  player_count++) {

                  /* increase our totals */
                  totals [player_count] +=
                          scores [game_count] [player_count] ;

                  /* decide if we have a highscore  */
                  /*  bigger than our current one */
                  if ( scores [game_count] [player_count] >
                          topscore) {
                           topscore =
                                  scores game_count][player_coun t] ;
                  }
         }
}

/* Now print out the results */
printf ( "\n\nScore totals:\n\n" ) ;
for (    player_count = 0 ;
         player_count < TEAMLENGTH ;
         player_count++ ) {
         printf ( "Total for player %d : %d \n", player_count+1,
                  totals [player_count] ) ;
}
printf ( "\n\nHighest score was : %d\n\n", topscore ) ;
}
```

You will find it very useful to study this code carefully, it contains many programming tricks you will use in your programs over and over again!

As a little exercise you may wish to try the following modifications to the program :

? print out the number of the player with the highest score.

? print out the number of the player with the highest average score.

? print out the total number of runs scored in the season.

Switching

Making Multiple Decisions

We now know nearly everything you need to know about constructing a program in the C language. You may find it rather surprising, but there is really very little left to know about programming itself. Most of the rest of C is concerned with making the business of programming simpler. A good example of this is the **switch** construction. Suppose you are refining your double glazing program to allow your customer to select from a pre-defined range of windows. You ask something like

Enter the type of window:

1 = casement
2 = standard
3 = patio door

Your program can then calculate the cost of the appropriate window by selecting type and giving the size.Each function asks the relevant questions and works out the price of that kind of item.

When you come to write the program you will probably end up with something like:

```
void handle_casement (void)
    ....
    .... definition o f handle_casement
    ....
void handle_standard (void)
    ....
    .... definition of handle_standard
    ....
void handle_patio (void)
    ....
    .... definition of handle_patio
    ....
void main (void)
{
    char select ;
    printf ("\n 1 = casement") ;
    printf ("\n 2 = standard") ;
    printf ("\n 3 = patio door") ;
    printf ("\nEnter the type of window : ") ;
    scanf ("%c", &selection) ;
    if (selection == '1') {
```

```
            handle_casement () ;
    }
    if (selection == '2') {
            handle_standard () ;
    }
    if (selection == '3') {
            handle_patio () ;
    }
}
```

This would work OK, but is rather clumsy. You have to write a large number of **if** constructions to activate each option.

Because you have to do this a lot C contains a special construction to allow you to select one option from a number of them based on a particular value. This is called the *switch* construction. If you write the above using it your program would look like this.

```
void main (void)
    char selection ;
    printf ("\nEnter the type of window : ") ;
    scanf ("%c", &selection) ;
    switch (selection){
    case '1' : handle_casement ();
            break ;
    case '2' : handle_standard () ;
            break ;
    case '3' : handle_patio () ;
            break ;
}
```

The **switch** command take a value which it uses to decide which option to perform. It executes the **case** which matches the value of the **switch** variable. Of course this means that the type of the cases that you use must match the switch selection value although, in true C tradition, the compiler will not tell you if you get it wrong, it will just delight in doing the wrong thing! One other possible naughty is the use of a **float** type for the switch selection. This is not allowed: because floating point numbers cannot be held exactly it would sometimes be impossible to find a match.

The **break** statement after the call of the relevant routine is to stop the program running on and performing the code which follows. In the same way as you break out of a loop, when the **break** is reached the **switch** is finished and the program continues running at the statement after the switch. Note that unless you give a **break** C will continue to run down past all further cases, i.e. the case item just says where to start running in the switch, **not** where to stop. Consider the following :

```
int i ;
i = 1 ;
switch ( i ) {
    case 1 :
            printf ( "one\n" ) ;
    case 2 :
            printf ( "two"\n ) ;
    break ;
}
```

Because there is no **break** at the end of the handling of the case for 1, the program would print out :

one
two

You can use this to good effect if you want a certain case to be selected by more than one value of the switch variable :

```
case 'c' :
case 'C' :
case '1' :
    handle_casement () ;
    break ;
```

This would cause handle_casement to be called if the user pressed 'C', 'c' or '1'. You can have as many additional options as you like.

Another other useful feature is the **default** option. This gives the switch somewhere to go if the switch value doesn't match any of the cases available; in our case (sorry!) we put out an appropriate message, for example :

```
void main ()
    char selection ;
    printf ("\nEnter the type of window : ")  ;
    scanf ("%c", &selection) ;
    switch (selection){
    case '1' : handle_casement ();
            break ;
    case '2' : handle_standard () ;
            break ;
    case '3' : handle_patio () ;
            break ;
    default : printf ("\nInvalid command\n")
}
```

The program would print out "**Invalid Command**" if anything other than a value with a matching case was entered.

Strings

How long is a piece of string?

Strings are computer jargon for lumps of text. The computer itself can get by quite happily with numbers, but we fuddy duddy old humans seem to prefer chunks of text, for example I prefer to be referred to as Rob Miles rather than **0883059276**! If you want to write a program which refers to someone by their name, rather than some meaningless numbers, you need to have a mechanism for storing it. Some programming languages have string handling built into them. The best example is BASIC, which contains special instructions for

string manipulation. C is not like BASIC. In C you have to set up strings yourself, the hard way. At first this is more tedious, but you find that the C way of doing things is much more flexible.

So, what do we mean by string? A string is any sequence of characters. A character is the kind of thing you get from a single keypress on the keyboard, or the thing you see in one position on the screen. We already know about the C data type called **char**, you can regard a string as a series of characters. In C we have just seen that the way to get yourself a series of memory locations is to declare an array of them. This means that in C strings and arrays of characters are exactly the same thing:

> **char name [20] ;**

The above would declare a string capable of holding a 20 character name.

There is just one more thing that you need to know about strings. Suppose I put my name in the above string. I put the characters 'R', 'O', 'B' into the first three locations. Then I have a problem. The array has been set up to hold names up to 20 characters long. This is to allow for unfortunates called Murgatroyd, who need the space, but I do not. I need to have a way of telling C that this is the end of my name, and that the remaining spaces are not used.

C lets me do this by use of a special convention: All strings are terminated by a character holding the value **0**. To terminate my name, I simply put a **0** into the location after the last character. You could I suppose regard 0 as the Arnold Schwartznegger of characters! The **0** at the end of the string is often called a *null* character. Maybe we are back to Arnold Schwartznegger again! Do not confuse the null character with the character code which represents **'0'**. **0** is the character you get when you press **'0'** on the keyboard. It is represented by the character code **48**. Null is an internal value used by C.

This means that the data space after the terminator is unused by my program. This can be a problem, but is not worth losing much sleep over. Later on we will look at ways of getting exactly the amount of memory space that we want.

Remember that when you create a string to hold some text, you must allow space for the terminator as well, i.e. **name [20]** has room for 19 characters of the name, followed by the terminator. Do not make the mistake of saying, "Ah well, if the name is 20 characters I do not need the terminator, because C knows the array is only 20 characters long". This is not the case. If you miss the terminator off C will wander down memory looking for a null, and consequently think that your string is a lot longer than it actually is!

Putting Values into Strings

The fact that strings are arrays of characters with a null on the end is something which the C compiler already knows. We put string constants into our code by putting them between double quote characters :

> **"hello mum"**

When the compiler sees the first " it says "Aha! Here is a string. I will store it in memory, and when I seen the closing " I will put a null on the end. I will then create a pointer to this piece of text and use that in the program." Quite a mouthful huh? Note however that the pointer which is created is a constant. This means that you cannot change where it points. There is a good reason for this; if you did change the pointer you would then have a lump of memory which could not be accessed, because nothing would point to it. If you want to put starting values into strings (or any kind of array) you have to do something like this:

```
char name [20] = "Fred Bloggs" ;
```

You can use the same trick to put an initial value into any variable.

The declaration creates an array of characters, which you can regard as a string. True to the way that arrays work, name is a pointer to your area of memory. When the compiler processes the string "**Fred Bloggs**" it ends up creating a pointer to an area of memory with :

F r e d   B l o g g s null

in it. Because of the way C works, it is only possible to initialise arrays which are declared as global variables, i.e.

```
char setting [10] = "off" ;
```

is OK but

```
void message (void)
{
    char setting [10] = "off" ;
}
```

- would cause a compilation error because this time **setting** is a local variable. You can initialise simple local variables, i.e.

```
void message (void)
{
    int i = 99 ;
}
```

- would be OK.

Using Strings

As far as C is concerned, a string is simply an array of characters with a null on the end. You can use all the normal array and pointer operations on this chunk of memory. For example, consider the problem of taking a full name as above, and printing out only the surname portion.

The first thing we must decide is how we determine where the surname starts. The convention would seem to be that the surname starts after the space in the name. What we therefore have to do is print the name, starting from the character after the space. This means that the first task is to find the space in the name. We do this by searching down the array, starting at the beginning and looking for a space character. When we see one we stop. The following code will do this :

```
/* our array name will hold the name */
/* we will see how to read the name later */
char name [20] ;

/* position will hold the position in the name */
int position ;
position = 0 ;        /* look from the start */
while ( name [position] != ' ' ) {
    /* If not a space... */
    position++ ;
     /* ..move on to next */
}
/* When we get here position has the subscript value */
/* of the space in our string. */
```

The variable **position** contains the subscript of the space in our name. We want to start printing from the character after the space. We must therefore move on to the next location :

```
position_++ ;
/* Move down one */
```

OK. Now for the interesting bit. We will be giving **printf** a pointer to the character which we want to start printing at. We know that **name [position**] is the first character that we want to print out. We also know that **printf** is supplied with the position in memory to print from, and prints until it sees a null. What we therefore want to do is start printing from the position of the first character, up to the end of the string. We can do this in C with :

```
printf ( &name [position] ) ;
```

We can put an **&** in front of any variable to get the address of it. By giving **&name [position]** we are giving the address of the **'B'**. We know that after that character comes the surname, so the program will print out what we want. If you do not believe this, try running the program!

This piece of code is not perfect, you might like to consider what would happen if the user gave a name with no space it, or a name with several spaces between the first name and the surname. I would expect any of you to write programs which would worry about these things as a matter of course. This technique is called defensive programming and translates to "make sure the problem appears somewhere else". You should practice defensive techniques every time you write some code. Once you have worked out how to solve the problem you should then go back and wonder how your solution can go wrong, and add extra handling for that!

Note that you could have done the same job using pointers rather than array elements and subscripts. See if you can understand this :

```
/* name array as before */
char name [20] ;

/* points to the start of the surname */
char * SurnameStart  = name ;

while ( *SurnameStart != ' ' ) {
    SurnameStart++ ; /* skip to the space */
}
SurnameStart++ ; /* move past the space */
printf ( SurnameStart ) ;
/* print starting at the surname */
```

The code does exactly the same job, the difference is only in how it is expressed. You might like to consider which of the two programs is better and why.

The String Library

Unlike some other languages, for example BASIC, C does not have any string handling "built in". Instead, as for input/output, it relies on a set of library routines which are supplied with the C system. These routines are common across all versions of C, and are specified in the **string.h** header file. You can use these routines to do string copying, comparison and concatenation for you.

Here are a few routines you may find useful :

strcpy

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

String copy. Has two parameters, both of them pointers to char. Will copy characters from the source to the destination, up to the null terminator of the source:

char name [20] = "Fred Bloggs" ;
char safety [20] ;

strcpy (safety, name) ;

would result in **safety** holding the string "**Fred Bloggs**".

strcpy returns the number of characters that it transferred. Note that if you use **strcpy** with an un-terminated string on the input you will get big problems!

strcmp

int strcmp (char * s1, char * s2) ;

String compare. Compares one string with another, and returns the result 1 if the first string was greater than the second, 0 if the two strings are the same and -1 if the first string was less than the second. C uses the character codes of the strings to decide on greater and less than, meaning that normal rules of alphabetic ordering apply; i.e. a < b, A < a, 1 < A.

printf ("%d", strcmp ("Fred", "Jim")) ;

would print out **-1**, because **Fred** is less than **Jim** alphabetically.

strlen

int strlen (char * string) ;

string length. Used to find out the length of a string. You should be able to write a function to do this yourself, but like any good programmer you will always look for the easy way to do something.

printf ("%d", strlen("HelloMum")) ;

would print out **8**. (Note that the terminator is not counted as a character in the string.)

Reading and Printing Strings

You will often want to read strings from the keyboard, and print them out. C provides another format specifier, **%s** to mean a string. This means that you can read a string from the user with code which looks like this :

char YourName [50] ;
scanf ("%s", YourName) ;
printf ("Hello %s\n", YourName) ;

Note that because **YourName** is actually a pointer to a char (that's what arrays are) you do not need to put the **&** pointer in front of the name.

However, doing formatted reads with strings is not usually a good idea. C has in its little head the idea of white space. White space is a gap which marks the

end of one thing and the start of another. Things separated by white space are different values, for example

2   3

- is not the value **23**, but the value **2** followed by the value **3**, with white space in between. White space can be described as :

Any number of spaces or newlines or tab characters.

This means that the string :

Rob Miles

- is actually two strings, **Rob** followed by **Miles**. If you really want to read in a line of data which may contain white space you should use a new function called **gets** (getstring). **gets** is a routine in **stdio** which fetches characters until the end of a line, so you can read data containing spaces :

```
char buffer [120] ;
gets (buffer) ;
```

Note that gets does not check for the length of the string, it assumes that you will have reserved enough space to hold the text. The trick is to reserve the length of a terminal line plus a bit, I usually make such lines 120 characters long. If I wanted to hold the name in a smaller sized string I would copy it there once I had read it in, and this time I would check the length!

While we are on the subject of useful routines in **stdio** I will mention **getchar**. This allows you to read a single character from the keyboard, without the user having to press the return key. It returns the character pressed :

```
char ch ;
ch = getchar ();
```

Note that you have to put the **()** after the call of **getchar** so that the compiler can tell that this is a function we are calling.

Bomb Proof Input

Users are stupid. Really stupid. They find ways of crashing your programs which you would never think of in a million years. The experience of seeing some idiot with half a brain cell blow away your wonderful program with a single keypress is a very depressing one. You must always bear in mind that if someone crashes your program you look stupid and they look clever. Any programmer that says to the user "You idiot, you have crashed my program" is not a Real Programmer, he is just a pretender. What you should say is "Oh Dear, what keys did you press", and then take steps to ensure that it never happens again.

One of the places where things can go wrong is when your program innocently asks for a number :

```
int Age ;
printf ( "How old are you : " ) ;
scanf ( "%d", &Age ) ;
printf ( "You are %d years old.\n" ) ;
```

This is a perfectly legal piece of code, but what happens if your user types :

twenty five

- and presses return. Try it!

Who made the mistake, was it the programmer or the user? Since we have not got the time to discuss this, we should simply make sure that our program is proof against it, and says something like "**I did not recognise that value, please type in a value, e.g. 25, and press return** "

We do this by always regarding input from the user as a string. We read the string in and try to convert it to a number. If the conversion fails, we ask for another string.

To do this we use a very useful function in stdio called **sscanf**. This does all the things that **scanf** does, but takes the input from a string, rather than the keyboard. Like **scanf**, **sscanf** returns the number of items it successfully read, so that we know if the values made sense. Consider the following :

```
void main (void)
{
    char buffer [120] ;
    int value, result ;
    do {
            printf ( "Give me a value : " ) ;
            gets ( buffer ) ;
            result = sscanf ( buffer, "%d", &value ) ;
    } while (result != 1) ;
    printf ( "%d\n", value ) ;
}
```

If you run this program and give it some values you can only get out by giving a valid integer. In your programs you should always use this technique when reading numbers from users.

Structures

What is a Structure?

Often when you are dealing with information you will want to hold a collection of different things about a particular item. For example, consider if the Nat. West. Bank commissioned you to write a customer database system. (it is in fact rather unlikely that this will happen - such systems are written in COBOL, not C!). Like any good programmer who has been on my course you would start by doing the following:

1. Establish precisely the specification, i.e. get in written form exactly what they expect your system to do.

2. Negotiate an extortionate fee.

3. Consider how you will go about storing the data.

From your specification you know that the program must hold the following:

- ? customer name - 30 character string

- ? customer address - 60 character string

- ? account number - integer value

- ? account balance - integer value

- ? overdraft limit - integer value

The Nat. West. have told you that they will only be putting up to 50 people into your database so, after a while you come up with the following:

```
#define MAX_CUST 50
#define NAME_LENGTH 30
#define ADDR_LENGTH 60

char name [NAME_LENGTH] [MAX_CUST] ;
char address [ADDR_LENGTH] [MAX_CUST] ;
int account [MAX_CUST] ;
int balance [MAX_CUST] ;
int overdraft [MAX_CUST] ;
```

What we have is an array for each single piece of data we want to store about a particular customer. If we were talking about a database (which is actually what we are writing), the lump of data for each customer would be called a record and an individual part of that lump, for example the overdraft value, would be called a field. In our program we are working on the basis that **balance[1]** holds the balance of the first customer in our database, **overdraft [1]** holds the overdraft of the first customer, and so on.

This is all very well, and you could get a database system working with this data structure. However it would me much nicer to be able to lump your record together in a more definite way.

C lets you create data structures. A structure is a collection of C data types which you want to treat as a single entity. In C a lump of data would be called a structure and each part of it would be called a member. To help us with our bank database we could create a structure which could hold all the information about a customer :

```
struct customer
{
    char name [NAME_LENGTH] ;
    char address [ADDR_LENGTH] ;
     int account ;
    int balance ;
    int overdraft ;
} ;
```

This defines a structure, called **customer**, which contains all the required customer information. Having done this we can now define some variables :

```
struct customer OnlyOne ;
struct customer EveryOne [MAX_CUST] ;
```

The first declaration sets up a variable called **OnlyOne**, which can hold the information for a single customer. The second declaration sets up an entire array of customers, called **EveryOne** which can hold all the customers.

We refer to individual members of a structure by putting their name after the structured variable we are using with a . separating them, for example :

OnlyOne.account

- would refer to the integer member **account** in the structured variable **OnlyOne**. You can do this with elements of an array of structures too, so that :

EveryOne [25].name

- would be the pointer to the part of memory containing the name of customer **25**.

How Structures Work

Knowing how structures work makes using them a lot easier. We know that C is actually very simple minded about the way that it does things, this includes how it handles structures.

When C is given a struct definition it goes Aha! Here comes a structure. I will work out how much memory it needs and remember whereabouts in this chunk of memory each member starts. This means that for our bank example :

char name [30]   The first 30 locations of the structure hold the name.

char address [60]   The next 60 locations hold the address.

int account   Next comes space for the account number

.. and so on. When you declare a variable of this structured type C makes a variable of the correct size. When you then do something like

OnlyOne.account = 99 ;

C says, I remember that the **account** value is stored after the **90** characters of the **name** and **address** members, and puts the value in the correct place.

If you have an array of structured variables, C just makes a variable with is :

size of array * size of structure

- in size and then fills in the members of each structure as it goes.

This means that, if you know how much memory **int**, **float** and **char** variables take up, you can work out how much memory any given structure needs. However, since C has already done this for you, it seems a little silly to have to do it manually. C provides a facility called **sizeof (item)**. This returns the number of locations which the item takes up in memory. We could do things like :

printf ("%d locations.\n", sizeof (struct customer)) ;

It is very useful to be able to do this, particularly when you start sending the contents of structures to files.

Pointers to structures

You can have pointers to structured data types, just like you can point to any other type, for example

struct customer * cust_pointer ;

This is a pointer to a customer record. You would set it to point somewhere in the usual way :

cust_pointer = &OnlyOne ;

If you think about it, we must be able to use pointers to structures because we can have arrays of structures and we know that an array is simply a lump of memory of the appropriate size, with a pointer to the bottom of it.

C keeps track of pointer types for us. This is particularly important when we start modifying the value of a pointer, for example :

cust_pointer++ ;

This means increase the value of **cust_pointer** by one, i.e. make **cust_pointer** point to the next customer record in memory. C knows how many memory locations that a customer record takes, so it moves down memory that amount.

When we have used a pointer before we have used a ***** to de-reference it. De-reference means get me the thing that this pointer points at so that :

*cust_pointer

the -> is made up of the character minus (-) directly followed by the character greater than (>)

- means the structure pointed at by **cust_pointer**. When you want to get hold of one of the members of a structure which you are pointing at you have to tell C to de-reference the pointer to the structure and then return the required member. You do this by replacing the **.** between a structured variable and a member name with a -> between the pointer and the member name :

cust_pointer->balance = 0 ;

- would set the balance of the customer pointed at by **cust_pointer** to **0**.

Defining your own Types

We have created our own types above, we can use them in our programs to hold specific kinds of data. However, we have to go through all the rigmarole of putting struct customer each time we want to declare a variable of that type. Also, at other points in the program we may want to use particular data types which mean something to us, how about a data type called inches which allows us to store values of inches. The reasons for doing this are not to make programs faster or smaller, but to make them easier to understand.

You can create a type using the new C keyword, **typedef**. This is followed by the type you want to declare :

typedef int inches ;

- this creates a new type called inches which is exactly the same as int. This means that we can put things in our program like :

inches width, height ;

- which declares **width** and **height** as variables of this type. This is a trivial, though useful, example. What we really want to do is create a new type which can hold our bank information:

```
typedef struct
{
    char name [NAME_LENGTH] ;
    char address [ADDR_LENGTH] ;
    int account ;
    int balance ;
    int overdraft ;
} customer ;
```

Note that we have moved the name of the type we are creating to the end, and put the **typdef** keyword at the start.

Now we have a new type, called **customer** which we can use just like any other type :

```
customer Hull_Branch [100] ;
customer *New_Customer_Pointer ;
```

The structures above are used in just the same way as the ones we declared earlier, but their declaration is much simpler.

Files

When do we use Files?

If you want your program to be properly useful you have to give it a way of storing data when it is not running. We know that you can store data in this way, that is how we have kept all the programs we have created so far, in files.

Files are looked after by the operating system of the computer. What we want to do is use C to tell the operating system to create files and let us access them. The good news is that although different operating systems use different ways to look after their files, the way in which you manipulate files in C is the same for any computer. We can write a program which creates a file on a PC and then use that program to create a file on a UNIX system, with no problems.

Streams and Files

C makes use of a thing called a *stream*. A stream is a link between your program and a file. Data can flow up or down your stream, so that streams can be used to read and write to files. The stream is the thing that links your program with the operating system of the computer you are using. The operating system actually does the work, and the C system you are using will convert your request to use streams into instructions for the operating system you are using at the time:
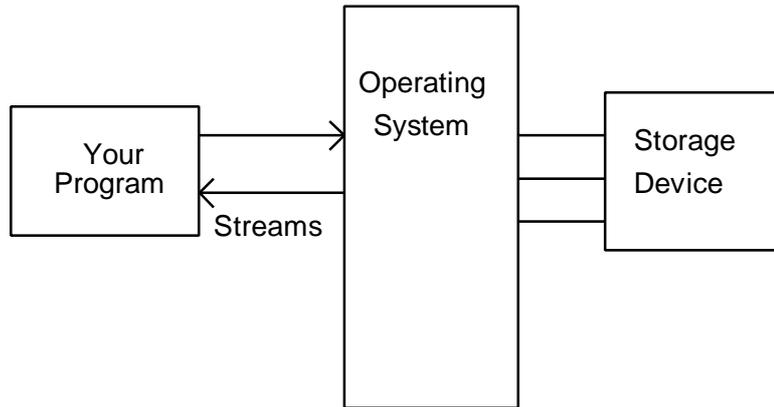
C needs somewhere to keep track of a particular stream, it needs to be able to remember where you have got to in the file, what you are using the file for and so on. We do not need to know just what information C stores about each file, and this information may well be different for each operating system.

C hides all this from us by letting us talk in terms of a structure called a **FILE**. A file is a structure which holds information about a particular stream. We do not create or manipulate this structure, that is done by the input and output routines that come with our version of C. All we have to do is maintain a pointer to a **FILE**, so we can tell the C functions which we want to use. If you are really interested, you can find out what a **FILE** is made of by looking in the file **STDIO.H**.

All the functions to manipulate files are defined in the **STDIO.H** file. They look very similar to the printf and scanf routines that we have used already. To remind you that they operate with files, all the file handling function names begin with "f".

fopen and fclose

The first step in using a file is to open it. Remember that a file can be used in different ways; for reading from or writing to, or perhaps both. C lets us protect files that we only want to read from by allowing us to open a file in read mode. This means that we are not allowed to change the contents of the file opened for reading. You tell the input/output system about the file you want to open by means of a mode string. This gives information about the type of file you are working on and the way in which it is to be used.

We open the file by using the function **fopen**. This has two parameters; the name of the file to be opened and the mode to use. It returns a pointer to a **FILE** structure which it creates, for example :

```
FILE * listing_output ;
FILE * program_input ;
listing_output = fopen ( "LISTING", "w" ) ;
program_input = fopen ( "PROGRAM", "r" ) ;
```

This opens a couple of files, **LISTING** is opened for output and **PROGRAM** is opened for reading.

See the section on POINTERS for more about NULL.

If the files do not exist, or there is a problem opening them, **fopen** returns a special value called **NULL**. You must always make sure that the open has worked before you try to do anything with a file, e.g.

```
FILE * listing_output ;
listing_output = fopen ( "LISTING", "w" ) ;
if (listing_output == NULL ) {
    printf ( "I could not open your output file. \n" ) ;
}
```

You can use a string as the name of the file you want to open, so that you could ask the user for the name of the output file and then open a file with that name. Note that the file is opened in the current directory, and that you must observe the conventions of the computer you are using with regard to file names.

When you open a file for reading, you are assuming that the file exists. If it does not, **fopen** will give you an error. When you open a file for writing the file may or may not exist, and **fopen** will create one for you if it needs to. This can lead to problems, what we have here is a very good way of destroying data by mistake. When you open an existing file for writing, even if the file is enormous, **fopen** will clobber the contents and start writing at the top of the file. You do not get a warning, all you get is a sinking feeling as several weeks work go down the tubes.....

Because I make a point of writing user-proof programs I therefore make sure that a user really wants to overwrite a file before I let him or her do it. This means that I find out if a file exists before I let the user write all over it. It is very easy to do this; you simply try to open the file for reading :

```
FILE * output_file ;
char output_file_name [50] ;
char reply [20] ;
do {
    printf ( "Give the name of your output file : " ) ;
    gets ( output_file_name ) ;
    output_file = fopen ( output_file_name, "r" )  ;
    if ( output_file != NULL ) {
            fclose ( output_file ) ;
            printf ( "Overwrite this file ? (Y or N) : " ) ;
            gets ( reply ) ;
            if ( reply [0] != 'Y' ) {
                    continue ;
            }
    }
} while ( ( output_file = fopen ( output_file_name, "w" ) ) ==
            NULL) ;
```

This snippet of code will loop until the user gives us a file which they say we can overwrite, and the file is opened successfully. Note the use of the underrated **continue** half way down. This causes the loop to start again from the top, which makes the program ask for another filename. Note also the use of the function **fclose**. **fclose** causes the file to be closed. It has one parameter, the file pointer whose file needs to be closed.

You must always close a file when you have finished with it. This is particularly important if you are writing to the file. The operating system does not switch on the disk drive to write just a single character to the disk, rather it waits until it has a load to write and then writes the lot in one go. This increases efficiency, but it does mean that at any time during your output some of the data is on the disk and the rest is in a buffer. Only when you call **fclose** is the buffer emptied and the disk written with all the information. If you want to force the buffer to be emptied onto the disk at any time, and ensure that it is up to date - but do not want to close the file, there is a function called **fflush** which will do this for you.

Mode String

The **fopen** function is told how to open the file by means of the mode string. The file opened is marked with the mode which is being used, and then other file input-output functions look at the mode before doing anything. This is how C stops you from writing to a file which was opened for input. The mode string can contain the following characters.

char	
w	open the file for writing. If the file does not exist it is created. If the file does exist it is emptied, and we start with a new one.
	Opens using w fail if the operating system is unable to open a file for output. This would usually be because the disk we are using is write protected or full, or if you are on a system which can share files, and somebody else has connected their program to the file in question.
r	open the file for reading. You will be unable to write into the file, but can read from it.
	Opens using r fail if the file does not exist, or if the file is protected in some way which denies us access.
a	open the file for append. If the file exists we are moved to the end of the file, i.e. if we send any data to a file opened for append it will be placed on the end of the file. If the file does not exist it is created for us.
	Opens using a generally fail for the same reasons as opens using w.
b	open the file as a binary file. Essentially there are two kinds of data on a computer. Stuff which makes sense to us, and stuff which makes sense to the machine itself. Stuff which makes sense to us is in the form of text, i.e. nothing in the file other than letters and numbers etc. Stuff which makes sense to the computer includes program files and any data file which needs to be translated by a program before it can be understood by people, e.g. spreadsheet data files. This data is called binary.
	If you open a file of type binary you are telling C that you want it to send the data exactly as you output it. i.e. it must not perform any translations which make this file easier for humans to make sense of.
t	open the file as a text file. Text files only contain printable characters, i.e. things that you or I would like to see. C will therefore make some changes to the file when it is output, usually in terms of what happens at the end of a line of text: some computer systems use two characters to mark the end of a line and others only use one.
	When a text file is opened the C input/output routines will perform the translation required. If you open a binary file as a text file you will notice very strange things happen. Remember that the C input/output system has no way of knowing which kind of file you really want to use, and so will do the wrong thing if you tell it to.
+	This means that you want to use the file for both reading and writing. You can put + after r or w. If you put it after a r (read) it means that the file is not destroyed if it exists, and that an error is produced if the file does not exist. If you put + after a w (write) it means that the file is destroyed if it exists, and a new file is created if required.

Some examples of file open modes and what they mean :

"rb+"   Open the file for reading and writing in binary mode. If the file does not exist do not create it. If the file does exist do not destroy it.

"wt"   Open the file for writing in text mode. If the file does exist destroy it. If the file does not exist create one.

File Functions

As I said above, we can use the functions **fprintf** and **fscanf** to communicate with our files :

```
fprintf ( listing_output, "This is a listing file \n" ) ;
fscanf ( program_input, "%d", &counter ) ;
```

The functions work in exactly the same way as **scanf** and **printf**, except that they use the file linked to the **FILE** pointer rather than the keyboard and screen. There are file versions of all the input and output functions we have covered so far.

In addition there are some very useful functions which let us save and load chunks of memory in files. These are very useful when you come to put structures and arrays into files. You might think that to save an array to disk you have to write each individual element out using some kind of loop. You can do this, but there is a much easier way of doing it.

fread and fwrite

You can regard an array, or an array of structured variables, as simply a block of memory of a particular size. The input/output routines provide you with routines that you can call to drop a chunk of memory onto a disk file with a single function call. However, there is one thing you must remember. If you want to store blocks of memory in this way the file must be opened as a binary file. What you are doing is putting a piece of the program memory out on the disk. You do not know how this memory is organised, and will never want to look at this, so you must open the file as a binary file. If you want to print or read text you use the **fprintf** or **scanf** functions.

The function **fwrite** sends a block of memory out to the specified file. To do this it needs to know three things; where the memory starts, how much memory to send, and the file to send the memory to. The location of the memory is just a simple pointer, the destination is a pointer to a **FILE** which has been opened previously, the amount of memory to send is given in two parts, the size of each chunk, and the number of chunks. This might seem rather long winded, but is actually rather sensible. Consider :

```
typedef struct
{
    char name [30] ;
    char address [60] ;
    int account ;
    int balance ;
    int overdraft ;
} customer ;

customer Hull_Branch [100] ;
FILE * Hull_File ;
    .
    .
    .
fwrite ( Hull_Branch, sizeof ( customer ), 100, Hull_File ) ;
```

The first parameter to **fwrite** is the pointer to the base of the array of our customers. The second parameter is the size of each block to save. We can use **sizeof** to find out the size of the structure. The third parameter is the number of customer records to store, in this case 100 and the final parameter is the file

which we have opened. Note that if we change the number of items in the customer structure this code will still work, because the amount of data saved changes as well.

The opposite of **fwrite** is **fread**. This works in exactly the same way, but fetches data from the file and puts it into memory:

```
fread ( Hull_Branch, sizeof ( customer ), 100, Hull_Data ) ;
```

If you want to check that things worked, these functions return the number of items that they transferred successfully :

```
if ( fread ( Hull_Branch, sizeof ( customer ), 100, Hull_Data )
     < 100 ) {

    printf ( "Not all data has been read!\n\n" ) ;

}
```

The End of the File and Errors

Any file maintained by the operating system has a particular size. As you write a file it is made bigger until you close it, so the only problems that you have when sending output to a file are concerned with what happens when you run out of disk space. We have already seen above that the standard C input/output functions can tell you how many items they have successfully transferred; you should always use the value they give back to test that your dealings with files are going properly.

When you are reading from a file it is useful to be able to check whether or not you have reached the end, without just failing to read. The function **feof** can be used to find out if you have hit the end :

```
if ( feof (input_file ) ) printf ( "BANG! \n" ) ;
```

This function returns the value 0 if the end of the file has not been reached, and another value if it has. It takes a **FILE** pointer as a parameter.

Note that binary and text files have different methods of determining the end of a file. If you are having problems because you keep reaching the end of the file before you think you should it may be because you have opened the file in the wrong mode.

feof has a twin brother called **ferror** who can be called to find out if an error has been caused due to a file operation. It is worth calling this if you find that less items have been transferred by a read or a write than you expected. The error number that you get back is specific to the operating system you are using but it can be used to make program more friendly, for example your program could tell the difference between "no disk in the drive" and "the disk has completely failed".

Memory

Fetching Memory

Sometimes you will come across situations where you do not know when the program runs exactly how much memory your program will need, for example you may want the user to tell you how many items he or she wants to store, and then get a chunk of memory just the right size. This is the most efficient way of writing your programs, it means that you do not use more resources than required. When a C program runs, a number of services are made available to it by the system it is running on. One of these is a memory manager. The memory manager looks after what memory is used by who. On a single user system, like the IBM PC, this is a very simple affair, either you have the memory or the system does. On a multi-user system, like UNIX, things are a lot more complicated. You however do not need to worry about this, you simply ask the memory manager for a chunk of memory and it will give you it if there is enough left. When you have finished with the memory it is nice if you give it back, so that other programs can use it.

The include file **alloc.h** contains all the memory allocation routines. You can find out how much memory is left, the size of the biggest block available and things like that. We are only going to use two, **malloc** and **free**.

malloc

This function will try to get a chunk of memory for you. It returns a pointer to the memory it found, if it could get some. If not it returns a **NULL** pointer. You must always check to make sure that malloc worked before you use the memory you think you got! **malloc** has one parameter, the size of the block of memory you want. You can use **sizeof** to find out how much memory you need, for example :

```
sizeof ( int ) * 99
```

- would be enough space for **99** integers. Because **malloc** returns a pointer to an item of type **void** you must use the cast facility to make this compatible with the type you really want; i.e. if you want to store **99** integers you will have a pointer to integers which you want to make point at the block of memory which malloc is going to get for you. The cast will make sure that the compiler does not produce an error when it sees the assignment, for example :

```
int * buffer ;
buffer = (int *) malloc ( sizeof ( int ) * 99 ) ;
if ( buffer == NULL ) {
    printf ( "No memory!\n" ) ;
}
else        {
    ......
    do something with the memory...
    ......
}
```

I would access an item in **buffer** in exactly the same way as I would access elements in any block of memory or array, for example **buffer [20]**. Note that the effects of going over the end of a block of memory which has been fetched in this way as just the same as any other - your program will do stupid things and then fall over! Note also that you have no idea what is in the memory when you get it! Do not assume that it has been emptied of silly values. There is a function, **calloc**, which you can use if you want all your memory to be cleared before you get it. This function is used differently from **malloc**, so you will have to look it up.

free

When you have finished with some memory you can give it back to the memory manager by the use of the **free** function, for example :

free (buffer) ;

This would give back the memory we requested above. Note that if we try to use this memory again, it may have been used for something else and so this would cause big problems. It is good housekeeping to always give back memory which you have allocated. If you do not you might find that the total amount of memory available will drop to a point where your program will not run any more.

The heap

The area of memory which is used for **malloc** and **free** is often called the heap. The name is apt, there is no organisation of it so after a while it becomes a mess. This can be a problem; what usually happens is that the heap becomes fragmented. This means that there is a lot of free space, but it is spread all over the heap. Your program can then fail to get the memory it wants, not because there is insufficient available, but because there is not a single chunk of the required size. You can guard against this happening by not allocating and freeing lots of little bits of memory repeatedly. Try to work by just grabbing a chunk and then holding on to it. Do not worry about giving it back if you know you will need it later.

If you want to be totally selfish, but also bomb proof, grab a very large amount of memory at the start and then manage it yourself. The advantage of doing it this way is that you can be sure when your program starts that it will have enough memory to run right to the end.

C and Large Programs

Building Large Programs in C

One area where C scores highly is in the development of very large programs. In C there is a standard way of performing separate compilation, we have already seen it in action when we considered how input and output is performed.

You can split your large C program into several smaller files. Each file will contain a number of C functions which do some parts of the project. In a large development different programmers will be working on particular files in the system, this is quite easy to manage in C.

The Compile and Link Process

When you want to build your working program you must compile each source file and then link them all together. The compiler does not produce the finished program, instead it produces an intermediate file which contains the machine code for a particular source file along with details of the names of the variables used and the names of the functions compiled. This is usually called the object file. C uses the file extension facility provided by the operating system to tell the files apart:

MENU.C   the .C file extension means that this is a source file which contains the C program text.

MENU.OBJ   the .OBJ file extension means that this is a file containing the compiler output for MENU.C. You cannot run this program, it must be processed by the linker to turn it into a binary program which will run.

      You may have several object files for a large project, each of which was produced from a single source file. These are fitted together, along with the library code, by the linker which produces:

MENU.EXE   the .EXE file extension means that this file contains a binary program which can be loaded and run by the operating system.

Note that these language extensions are the ones which MS-DOS uses. If you use UNIX the extensions are different, but they are used to the same effect.
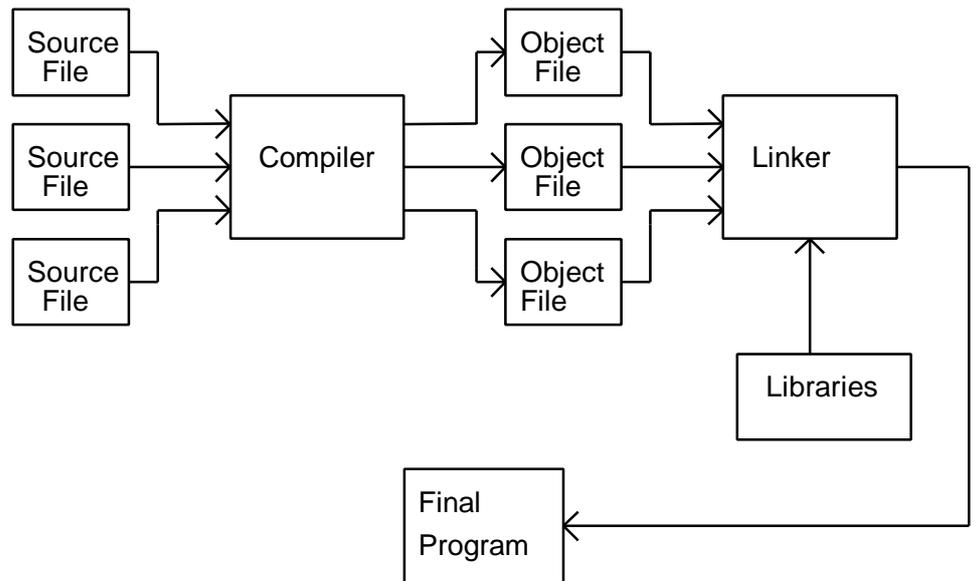
The linker ties up all the separate files, for example if one program refers to a function called setup the linker will convert this reference to a call to setup defined in another file. Note that if the linker does not find a function called setup the linking process fails with an error.

You get linker errors if you refer to functions that do not exist, or define a function with the same name as one somewhere else. This means that even if your program compiles OK, it may still have errors in the text if you have spelt a function incorrectly.

Referring to External Items

If you are writing one file which is part of a large system, you will want to refer to other functions which are not local to your part. (We are already doing this when we use **scanf** and **printf**.) In the case of these routines there is a standard file called **stdio.h** which contains the definitions of external functions we want to refer to. We can build our own file of such definitions if we wish, and use them to refer to distant code. C will let you set up a function prototype. This is just the top part of a function, analogous to the forward declaration facility of PASCAL. All the compiler needs to know about an external routine is what it looks like, i.e. the name, the type of value returned by it and the number and type of the parameters. That is what the prototype gives it:

```
void increment ( int * it ) ;
```

This is a prototype for the **increment** function we wrote earlier. When C sees this it just drops a reference to that function into the object file and then expects the linker to sort things out when it builds the final program.

If I want to refer to external variables in my file I must tell C what they are called and what type they are. I can do this using the extern declaration modifier:

```
extern int i ;
```

This says to the compiler there is a variable called **i**, which is of type integer defined somewhere else. I do not want you do set up a variable of that name, just pretend that one exists and let the linker sort things out!

If the project is very large you may have lots of separate files, each containing functions and variables that you want to share. There is a standard way of defining these things, we have already used it with the standard input/output definition file **stdio.h**. For each file which I want to refer to in other ones I create a ".**H**" file. This contains all the function prototypes and external variable definitions for other programs to use. If other files want to make use of these routines they simply have to **#include** this specification file, i.e.:

MENU.C   holds all the menu functions.

MENU.H   holds function prototypes and external variables which can be used to access the menu functions.

This is very useful, someone can use my menu routines without having to look at the actual code - the ".**H**" file contains all they need to refer to them.

If there were many programmers working on a large project the first things that they would write would be all the ".**H**" files which define how all the functions will fit together, each programmer can then go on and write the code to do his or her particular part and the others can use it without ever seeing it!

The other thing that you can put into your ".**H**" files is the design of any data structures that you are using. If your big project contains customised structures you might have a file called **STRUCTS.H** which everyone uses. This means that you are all using the same copy of the definitions.

The Make Program

Another effect of splitting your project up into a number of separate files is that it makes working on the system faster. If you change one of the files you need only re-compile that file and re-link in all the ones which have not changed. This is much better than having to re-compile everything. However it does bring another problem, that of keeping all your object files up to date and making sure that you do not use an out of date file at any time. Furthermore, if you are using vital ".H" files, you must re-compile those source files which use them if they are changed.

Doing all this manually is a bit of a pain, so instead C provides an automated make facility. This allows you to construct a project file which tells the make system the names of all the source files in the system, and which files they depend on. When you want to create a new version of your program you simply call the make program which reads this project file and then looks at the datestamps of all the source and object files. If any source file is newer than the corresponding object one it is re-compiled. Furthermore, if a file which has changed has other ones which depend on it, all the dependent files are re-compiled too.

Good versions of C have very powerful make systems, they are virtually a "programming language" which is used to specify how the application is to be built.

Projects

If you are using an integrated programming environment, for example Borland C or Microsoft C, you can also create projects, which are similar to make files but also allow you manage the files visually.

The C Pre-Processor

I have been using **#include** extensively throughout the examples. This is an instruction which tells the compiler to take the contents of a file and include it at that particular point in the program. This form of activity is actually handled by the C pre-processor. We have already looked at the pre-processor in the context of magic numbers and the **#define** directive.

The #include Directive

Pre-processor directives are preceded by a **#** sign and are the only thing on a line :

> **#include <stdio.h>**

When the pre-processor sees the **#include** directive it looks for a file with the name following it and opens that file. It then passes the contents of that file to the compiler. At the end of the include file it continues with the current source file. The file that you include can also contain **#include** directives and the pre-processor will nest them as required.

Note that we have enclosed the file name in **<>** characters. Enclosing the name in **<>** tells the pre-processor to look in a special system include area for the file. This is where standard definition files for all the C run time library routines are kept. If you want to tell the system to look in the local directory instead you use **""** to enclose the filename:

> **#include "menu.h"**

Conditional Compilation

You can get the pre-processor to selectively pass on parts of your program to the compiler. This is very useful when you are developing something and want to add additional debugging code. If you were using PASCAL you would have to remove or comment out all the debug statements when you produce the final version. In C you can just tell the pre-processor not to pass particular parts of the program to the compiler.

The decision is made depending on whether a particular symbol has been defined previously, for example:

> **#ifdef debug**
> **printf ("Intermediate value %d \n", i) ;**
> **#endif**

If **debug** had been defined previously the **printf** statement is passed to the compiler. If this symbol does not exist all the text between the **#ifdef** and the **#endif** is removed by the pre-processor and the compiler never sees it. This means that I can turn all my debug statements on simply by typing:

> **#define debug 1**

- at the beginning of the program and then rebuilding it. Note that the translation that I give to **debug** is not important, merely the fact that it has been defined.

You can add an else part if you wish:

```
#ifdef friendly
printf ( "Sorry, you made a mistake." ) ;
#else
printf ( "You idiot!" ) ;
#endif
```

Remember that these decisions are not made when the program runs, they actually control what the program actually contains.

Another popular use for conditional compilation is the building of code which can be customised for various different machines. A particular compiler often has a number of words pre-defined. Your source can check for these and then include code to customise the program for that particular version. This makes writing portable code a lot easier.

A Sample Project

This brings together some of the things I have been talking about above. We will use all of the techniques to make a large program out of a number of small files. We will also look at good design techniques and decisions you make when starting the project:

The Problem

We are writing a program that will keep track of orders. The data we want to store is:

?? the name of the customer

?? the address of the customer

?? the customer reference number

?? the name of the item purchased

?? the reference number of the item purchased

The program should allow orders to be entered, edited, search for and printed.

The Data Structure

By referring to the list above we come up with the following data structure to hold order information:

```
struct sale_record
{
  char customer_name [NAME_LENGTH] ;
  char customer_address [ADDRESS_LENGTH] ;
  int customer_ref ;
  int item_name [NAME_LENGTH] ;
  int item_ref ;
} ;
```

Our program is now concerned with the management of data held in these records.

We will create an array of these that will serve as the main data storage for our application:

```
#define DATABASE_LENGTH 50
struct sale_record sales [DATABASE_LENGTH] ;
```

Program Files

Once we have our data structure designed, we can start worrying about the programs that will manipulate it.

Rather than put all the code into a single file, we decide to spread the system over three source files:

?? user menu and displays - **menu.c**

?? data storage and management - **data.c**

?? data printing - **print.c**

Each source file will contain functions concerned with that part of the system, and the other files may make use of these. I have decided to put the **main** function in the **menu.c** source file. Taking each source file in turn:

data.h & data.c

The data portion of the system will look after all the storage and retrieval of data for our system. The **data.h** file must give enough information about the data storage organisation and facilities so that other source files can manage data.

Note that in order to maintain consistency I will **#include** the **data.h** file in the source file **data.c**. I do this so that the compiler can check for consistency between the function prototypes that I tell the other files about and the functions themselves in **data.c.** This does lead to a problem however in that some of the items in **data.h** will be **extern** declarations of variables which other files need to use. If I just include **data.h** in **data.c** I will refer to a variable as both extern and locally declared, leading to compiler errors.

To get around this I use conditional compilation. At the beginning of the file **data.c** I have the following line:

    #define data

This is then used inside **data.h** so that I only declare external variables if I am not inside **data.c**. I also use other tricks with **#define** to make sure that a given **#include** file is only included once, you can work out how these are used for yourself.

The complete **#include** file looks like this:

```
/*  include file for database management */

/*  handle multiple includes  */

#ifndef data_available

/*  set the lengths of our database items  */

#define NAME_LENGTH 100
#define ADDRESS_LENGTH 500

/*  define our database structure  */

struct sale_record
{
    char customer_name [NAME_LENGTH] ;
    char customer_address [ADDRESS_LENGTH] ;
    int customer_ref ;
```

```
        int item_name [NAME_LENGTH] ;
        int item_ref ;
} ;

/*  set the length of our database  */

#define DATABASE_LENGTH 50


/*  make an external reference to the database if we are not
inside data.c  */

#ifndef data

extern struct sale_record sales [DATABASE_LENGTH] ;

#endif

/*  Now put in the data management functions...  */

/*  values for the functions to return  */

#define TRUE 1
#define FALSE 0

int is_empty ( struct sale_record * this_record ) ;

void clear_record ( struct sale_record * this_reco rd ) ;

#define STORED_OK 0
#define NO_ROOM_IN_DATABASE 1

int store_record ( struct sale_record * this_record ) ;

struct save_record * find_record ( struct sale_record * pattern ) ;

#define SAVED_OK 0
#define SAVE_FILE_OPEN_FAILED 1
#define SAVE_FILE_FULL 2

int save_records ( void ) ;

#define LOADED_OK 0
#define LOAD_FILE_OPEN_FAILED 1
#define LOAD_FILE_SHORT 2

int load_records ( void ) ;

/*  tell other include files that structures are available  */

#define data_available

#endif
```

Note that the functions given are *function prototypes*, which describe what the functions are called and what they do. They are **not** the functions themselves. Taking some of the functions in detail:

I have created **TRUE** and **FALSE** so that functions which return something (for example whether they worked or not) have set values which can be picked up and used. If we look at one of the functions in detail::

int is_empty (struct sale_record * this_record) ;

This function returns **TRUE** or **FALSE**. If the record which **this_record** points at is empty, the function returns **TRUE**. If not it returns **FALSE.**

Note that the design of the name of the function would tend to let you know what it does, this is an example of *self-documenting code*.

The thing you may have bother with is:

> **struct sale_record * this_record**

This describes a pointer called **this_record** that is allowed to point at sale records. This way we can feed the function sale records to look at, for example:

```
struct sale_record x ;
if ( is_empty ( &x ) == TRUE ) {
    printf ( "x is empty\n" ) ;
}
```

If we want to look at one of the entries in our array we would put:

```
if ( is_empty ( &sales [25] ) == TRUE ) {
    printf ( "empty\n" ) ;
}
```

We will use the idea of pointers as parameters in a lot of the following functions. Giving a pointer to the thing we want to use is good for two reasons:

?? It allows the object to be changed by the function

?? It reduces the amount of data passed into the function, i.e. only the address of the object is passed, not the object itself

void clear_record (struct sale_record * this_record) ;

This function is used to make a record empty. We have to agree on what constitutes and empty record, perhaps we say that a record with an empty customer name is an empty one. Whatever method we use, we must make sure that the **is_empty** function agrees that this is empty. Note that this function doesn't return anything, because I can't think of a way in which it can fail (in fact I can - see if you can as well)

int store_record (struct sale_record * this_record) ;

I am implementing the database in such a way that the users of the **data.c** program don't need to know how everything is stored. They just give the data routings a pointer to a record that needs to be stored, and it goes ahead and stores it.

Note that this function can return whether or not the store request worked OK. I have #defined a couple of values to allow the function to indicate whether or not it worked.

struct find_record * find_record (struct sale_record * pattern) ;

This function searches for a record that matches a pattern given to it. We must construct the pattern and then we can use **find_record** to match this to each entry in the database in turn.

We will have to document how the pattern works, for example if there will be wild cards etc. If a match is found the function returns a pointer to the record. If a match is not found the function returns the **NULL** pointer.

int save_records (void) ;

This function is called when the database needs to be saved from memory into a file. Note that there are some error return values for this function to use.

int load_records (void) ;

This function is called when the database needs to be loaded into memory from a file. Note that there are some error return values for this function to use.

print.c & print.h

This set of functions manages the printing of the items in the database. Note that we use conditional compilation tricks to make sure that the database include file is around so that we can use it, and that we don't set things up if there is no need.

```
/*  just define the print  functions which we use  */

#ifndef print_available

/*  we will be using data structures, include for these if needed
*/

#ifndef data_available

#include "data.h"

#endif

#define PRINTED_OK 0
#define PRINT_FAILED 1

int print_record ( struct sale_recor d * this_record ) ;

#define DISPLAYED_OK 0
#define DISPLAY_FAILED 1

int display_record ( struct sale_record * this_record ) ;

#define print_available

#endif
```

You can probably figure out what each of the functions does, and what it returns in the way of error messages from the include file above.

menu.c

This is the main program. It uses the facilities provided by print and data to deliver what the user wants. Note that I don't have a **menu.h** file because no file makes use of functions in **menu.c** (this could change if I designed it diferently)

```
/*  main menu file  */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "data.h"
```

```c
#include "print.h"

int build_query ( struct sale_record * search_record )
{
    return TRUE ;
}

int read_record ( struct sale_record * this_record )
{
    return TRUE ;
}


void main ( void )
{
    char command ;
    int finished_program = FALSE ;

    if ( load_records () == FALSE ) {
        exit ( 1 ) ;
    }

    do {

        printf ( "Main Menu\n\n"
            "1 : Add Record\n"
            "2 : Delete Record\n"
            "3 : Search Records\n"
            "4 : Edit Record\n"
            "5 : Print Record\n"
            "6 : Exit Program\n\n"
            "Press command key : " ) ;

        command = getche () ;

        switch ( command ) {

        case '1' : /*  Add Record  */
        {
            struct sale_record temp_record ;
            if ( read_record ( &temp_record ) == TRUE ) {
                store_record ( &temp_record ) ;
            }
        }

            break ;

        case '2' : /*  Delete Record */

            break ;

        case '3' : /*  Search Records  */

            break ;

        case '4' : /*  Edit Record  */
```

```
                break ;

        case '5' : /*  Print Records  */

                break ;

        case '6' : /*  Quit the program  */

                finished_program = TRUE ;
                break ;

        default : /*  invalid command  */

                printf ( "Invalid command  \n\n" ) ;
                break ;
        }
    } while ( finished_program == FALSE ) ;

    save_records () ;

}
```

Please remember that this is probably not the best way to solve the given problem, it is simply the way that I do it!

The Most Important Bit!

This is the **most** important bit!

When you do your top level design, designing data structures, splitting the facilities across several source files and creating functions and values that they return it is very important that you document all this work. There should be an *implementation bible* that describes what each source file does and what the functions inside the file do. The best way to write this is as the work progresses. It should be made available to all the people working on the project (even if there is just you!) and it will form the first component of the software documentation, a "road map" of the project.

Later in the course we will look at formal design methods, for now I just want to make sure that you understand the importance of these aspects of software writing. When you solve a problem for a customer you are not just writing a program, you are creating a system which will solve the problem, and this will include consideration of the overall design and testing of the product you are going to create. It is very important that you take this "systems approach" from the beginning of your programming career!

Glossary of Terms

assembly language

Assembly language is the textual way of describing a machine code program. Each individual machine code instruction is represented by a letter sequence called a mnemonic. A program called an assembler converts the assembly language into machine code. You can mix assembler and compiler output by linking together their object files. You write assembler programs when you need great speed, or want to talk to particular pieces of hardware. Assembly language is slow to write and non-portable.

call

When you want to use a function, you call it. When a function is called the thread of execution switches to that function, starting at the first statement in its body. When the end of the function, or the return statement, is reached the thread of execution returns to the statement immediately following the function call.

compiler

A compiler takes a source file and makes sense of it. It is the first stage in the conversion of a program into machine code. The compiler will produce an object file which is linked in with other object files and library files to produce the machine code program which is run. Writing compilers is a specialised busines, they used to be written in assembly language but are now constructed in high level languages (like C!). A compiler is a large program which is specially written for a particular computer and programming language. Most compilers work in several phases. The first phase, the pre-processor, takes the source which the user has written and then performs all the compiler directives, producing a stream of program source which is fed to the "parser" which ensures that the source adheres to the grammar of the programming language in use. The final phase is the code generator, which produces the object file which is later linked by the linker.

directive

A directive is a command of some kind. In C this usually refers to the pre-processor, which acts on these to get particular effects. Some assemblers also support directives which can change the way they work, but these are not the same as the ones used in C.

---

format string

The formatted print and scan functions, for example scanf and printf, need to know how to format their output. To tell them the layout we use the format string. It contains characters which are to be transferred - for example hello, place markers for values - for example %i for an integer, and control sequences for layout - \n for a newline.

Functional Design Specification

Large software developments follow a particular path, from the initial meeting right up to when the product is handed over. The precise path followed depends on the nature of the job and the techniques in use at the developer; however, all developments must start with a description of what the system is to do. This is the most crucial item in the whole project, and is often called the Functional Design Specification, or FDS.

machine code

Machine Code is the language which the processor of the computer actually understands. It contains a number of very simple operations, for example move an item from the processor into memory, or add one to an item in the processor. Each particular range of computer processors has its own specific machine code, which means that machine code written for one kind of machine cannot be easily used on another.

object file

The object file contains the output of a compiler or an assembler and the names and types of variables used in that source file which the object was created from. It also contains references to things which were referred to in the source file but which were not present, for example library functions and external variables. The object file is acted on by the linker.

portable

When applied to computer software, the more portable something is the easier it is to move it onto a different type of computer. Computers contain different kinds of processors and operating systems which can only run programs specifically written for them. A portable application is one which can be transferred to a new processor or operating system with relative ease. High Level languages tend to be portable, machine code is much harder to transfer.

source file

You prepare a source file with a text editor of some kind. It is text which you want to pass through an assembler or a compiler to produce an object file for linking

test harness

The test harness will contain simulations of those portions of the input and output which the system being tested will use. You put your program into a test harness and then the program thinks it is in the completed system. A test harness is very useful when debugging as it removes the need for the complete system (for example a trawler!) when testing.

Index