About C

As a programming language, C is rather like Pascal or Fortran. Values are stored in variables. Programs are structured by defining and calling functions. Program flow is controlled using loops, if statements and function calls. Input and output can be directed to the terminal or to files. Related data can be stored together in arrays or structures.

Of the three languages, C allows the most precise control of input and output. C is also rather more terse than Fortran or Pascal. This can result in short efficient programs, where the programmer has made wise use of C's range of powerful operators. It also allows the programmer to produce programs which are impossible to understand.

Programmers who are familiar with the use of pointers (or indirect addressing, to use the correct term) will welcome the ease of use compared with some other languages. Undisciplined use of pointers can lead to errors which are very hard to trace. This course only deals with the simplest applications of pointers.

It is hoped that newcomers will find C a useful and friendly language. Care must be taken in using C. Many of the extra facilities which it offers can lead to extra types of programming error. You will have to learn to deal with these to successfully make the transition to being a C programmer.

C and UNIX

This course teaches C under the UNIX operating system. C programs will look similar under any other system (such as VMS or DOS), some other features will differ from system to system. In particular the method of compiling a program to produce a file of runnable code will be different on each system.

The UNIX system is itself written in C. In fact C was invented specifically to implement UNIX. All of the UNIX commands which you type, plus the other system facilities such as password checking, lineprinter queues or magnetic tape controllers are written in C.

In the course of the development of UNIX, hundreds of functions were written to give access to various facets of the system. These functions are available to the programmer in libraries. By writing in C and using the UNIX system libraries, very powerful system programs can be created. These libraries are less easy to access using other programming languages. C is therefore the natural language for writing UNIX system programs.

This Course

The course aims to introduce programmers to the C language. Previous programming experience is assumed, so we can quickly progress to looking at the features of C and their uses. Students with little programming experience will need to do some homework in order to keep up with the lectures.

Teaching will emphasise the use of supervised practical sessions, giving the student hands on programming experience. The student will collect a number of working practical programs which will be useful reference material for the future.

The notes will include examples and explanation as far as possible. We will try to avoid involved discussion of the syntax of the language. This subject is exhaustively covered in a range of books which are available from bookshops or the University Library.

We aim to introduce C in a structured manner, beginning with the simpler aspects of the language, and working up to more complex issues. Simple aspects will be dealt with rather quickly in order to leave more time for the more powerful features.

Common C

Until recently there was one dominant form of the C language. This was the native UNIX form, which for historical reasons is known as either Bell Labs C, after the most popular compiler, or K. &R. C, after the authors of the most popular textbook on the language. It is now often called "Classic C"

ANSI C

The American National Standards Institute defined a standard for C, eliminating much uncertainty about the exact syntax of the language. This newcomer, called ANSI C, proclaims itself the standard version of the language. As such it will inevitably overtake, and eventually replace common C.

ANSI C does incorporate a few improvements over the old common C. The main difference is in the grammar of the language. The form of function declarations has been changed making them rather more like Pascal procedures.

This course introduces ANSI C since it is supported by the SUN workstation compilers. Most C programming texts are now available in ANSI editions.

A Quick Overview of C

It is usual to start programming courses with a simple example program. This course is no exception.

- # A Very Simple Program
- This program which will print out the message This is a C program
- ```
  #include <stdio.h>
  ```
-
- ```
  main()
  {
          printf("This is a C program\n");
  }
  ```
- Though the program is very simple, a few points are worthy of note.

- Every C program contains a function called main. This is the start point of the program.
- #include <stdio.h> allows the program to interact with the screen, keyboard and filesystem of your computer. You will find it at the beginning of almost every C program.
- main() declares the start of the function, while the two curly brackets show the start and finish of the function. Curly brackets in C are used to group statements together as in a function, or in the body of a loop. Such a grouping is known as a compound statement or a block.
- printf("This is a C program\n");
  prints the words on the screen. The text to be printed is enclosed in double quotes. The \n at the end of the text tells the program to print a newline as part of the output.
- Most C programs are in lower case letters. You will usually find upper case letters used in preprocessor definitions (which will be discussed later) or inside quotes as parts of character strings. C is case sensitive, that is, it recognises a lower case letter and it's upper case equivalent as being different.
- While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. The following program will print a conversion table for weight in pounds (U.S.A. Measurement) to pounds and stones (Imperial Measurement) or Kilograms (International).

A Weight Conversion Program

```
#include <stdio.h>
#define KILOS_PER_POUND .45359
main()
{       int pounds;

        printf(" US lbs      UK st. lbs       INT Kg\n");

        for(pounds=10; pounds < 250; pounds+=10)
        {       int stones = pounds / 14;
                int uklbs = pounds % 14;
                float kilos = pounds * KILOS_PER_POUND;
                printf("   %d           %d   %d        %f\n",
                                pounds, stones, uklbs, kilos);
        }
}
```

Again notice that the only function is called main.

int pounds; Creates a variable of integer type called pounds.

float kilos; Creates a floating point variable (real number) called kilos.

#define KILOS_PER_POUND .45359
defines a constant called KILOS_PER_POUND. This definition will be true throughout the program. It is customary to use capital letters for the names of constants, since they are implemented by the C preprocessor.

```
for(pounds=10; pounds < 250; pounds+=10)
```
This is the start of the loop. All statements enclosed in the following curly brackets will be repeated. The loop definition contains three parts separated by semi-colons.

- The first is used to initialise variables when the loop is entered.
- The second is a check, when it proves false, the loop is exited.
- The third is a statement used to modify loop counters on each loop iteration after the first.

The effect of `pounds += 10` is to add 10 to the value of the variable pounds. This is a shorthand way of writing `pounds = pounds + 10`.

The printf statement now contains the symbols %d and %f. These are instructions to print out a decimal (integer) or floating (real) value. The values to be printed are listed after the closing quote of the printf statement. Note also that the printf statement has been split over 2 lines so it can fit onto our page. The computer can recognise this because all C statements end with a semicolon.

Weight Conversion Table Using a Function

The previous program could be better structured by defining a function to convert the weights and print out their values. It will then look like this.

```
#include <stdio.h>

void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
```

```
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

main()
{       int us_pounds;

        printf(" US lbs      UK st. lbs       INT Kg\n");

        for(us_pounds=10; us_pounds < 250; us_pounds+=10)
                print_converted(us_pounds);
}
```

void print_converted(int pounds) is the beginning of the function definition. The line within the loop reading print_converted(us_pounds) is a call to that function. When execution of the main function reaches that call, print_converted is executed, after which control returns to main.

The text enclosed by symbols /* and */ is a comment. These are C's way of separating plain text comments from the body of the program. It is usually good practice to have a short comment to explain the purpose of each function.

Defining a function has made this program larger, but what have we gained? The structure has been improved. This may make little difference to the readability of such a small program. In a larger program, such structuring makes the program shorter, easier to read, and simplifies future maintenance of the program. Another benefit of defining a function, is that the function can easily be re-used as part of another program.

Weight Conversion with a Prompt

To illustrate this, our next program will re-use the function from 2.3. The program is similar to the last one, but instead of printing a table of weights, the user enters a weight, and this value is converted. Such a program requires user input. This can be done in two ways, both of which will be shown.

The simpler implementation is to prompt for a weight, and then read the user's keyboard input. This would be done as follows

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

```
void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

main()
{       int us_pounds;

        printf("Give an integer weight in Pounds : ");
        scanf("%d", &us_pounds);

        printf(" US lbs      UK st. lbs       INT Kg\n");
        print_converted(us_pounds);
}
```

A printf statement is used to prompt for input. scanf is an equivalent input statement, note that the variable to be read us_pounds is written as &us_pounds here. This is very important and it will be explained later.

Weight Conversion with Command Line Argument

In this example, the number to be converted is supplied as part of the command to run the program. This can be a useful way of supplying a limited number of names or numbers to a program. We can run the program by typing program 1200 to convert 1200 lbs.

```
#include <stdio.h>

void print_converted(int pounds)
/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
{       int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos_per_pound = 0.45359;
        float kilos = pounds * kilos_per_pound;

        printf("   %3d          %2d  %2d        %6.2f\n",
                pounds, stones, uklbs, kilos);
}

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

```
        if(argc != 2)
        {        printf("Usage: convert weight_in_pounds\n");
                exit(1);
        }

        sscanf(argv[1], "%d", &pounds); /* Convert String to int */

        printf(" US lbs      UK st. lbs       INT Kg\n");
        print_converted(pounds);
}
```

The main function definition has changed so that it takes two arguments, argc is a count of the number of arguments, and argv is an array of strings containing each of the arguments. The system creates these when the program is run.

Some other new concepts are introduced here.

if(argc != 2) Checks that the typed command has two elements, the command name and the weight in pounds.

exit(1); Leave the program. The argument 1 is a way of telling the operating system that an error has occurred. A 0 would be used for a successful exit.

sscanf(argv[1], "%d", &pounds)
Converts a string like "100" into an integer value stored in pounds. The argument is stored in argv[1] as a string. sscanf works like scanf, but reads from a string instead of from the terminal.

The rest of the program is the same as the previous one.

Fibonacci Series Using an Array

We have now used a variety the features of C. This final example will introduce the array. The program prints out a table of Fibonacci numbers. These are defined by a series in which any element is the sum of the previous two elements. This program stores the series in an array, and after calculating it, prints the numbers out as a table.

```
#include <stdio.h>

main()
{       int fib[24];
        int i;
```

```
        fib[0] = 0;
        fib[1] = 1;

        for(i = 2; i < 24; i++)
                fib[i] = fib[i-1] + fib[i-2];

        for (i = 0; i < 24; i++)
                printf("%3d   %6d\n", i, fib[i]);
}
```

One new construct has been introduced here.

int fib[24];
This defines an array called fib of 24 integers. In C, the array index is bounded by square brackets (this avoids confusion with a function call). Also, the first element of the array has index zero, and for this 24 element array, the last element is index 23.

Following this brief scan through the language, we shall introduce the components of C in rather more detail.

Using C with UNIX

A little knowledge is necessary before you can write and compile programs on the UNIX system. Every programmer goes through the same three step cycle.

1. Writing the program into a file
2. Compiling the program
3. Running the program

During program development, the programmer may repeat this cycle many times, refining, testing and debugging a program until a satisfactory result is achieved. The UNIX commands for each step are discussed below.

Writing the Program

UNIX expects you to store your program in a file whose name ends in .c This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. To edit a file called testprog.c using vi type

```
vi testprog.c
```

The editor is also used to make subsequent changes to the program.

Compiling the Program

There are a number of ways to achieve this, though all of them eventually rely on the compiler (called cc on our system).

The C Compiler (cc)

The simplest method is to type

```
cc testprog.c
```

This will try to compile testprog.c, and, if successful, will produce a runnable file called a.out. If you want to give the runnable file a better name you can type

```
cc testprog.c -o testprog
```

This will compile testprog.c, creating runnable file testprog.

Make, a Program Builder

UNIX also includes a very useful program called make. Make allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called Makefile). If your C program is a single file, you can usually use make by simply typing

```
make testprog
```

This will compile testprog.c and put the executable code in testprog.

Improved Type Checking Using Lint

The C compiler is rather liberal about type checking function arguments, it doesn't check bounds of array indices. There is a stricter checker called lint which won't generate any runnable code. It is a good idea to use lint to check your programs before they are completed. This is done by typing

```
lint testprog.c
```

Lint is very good at detecting errors which cause programs to crash at run time. However, lint is very fussy, and generally produces a long list of messages about minor problems with the program. Many of these will be quite harmless. Experience will teach you to distinguish the important messages from those which can be ignored.

Running the Program

To run a program under UNIX you simply type in the filename. So to run program testprog, you would type

```
testprog
```

or if this fails to work, you could type

```
./testprog
```

You will see your prompt again after the program is done.

Constant and Variable Types

- # Variables
- In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.
- A declaration begins with the type, followed by the name of one or more variables. For example,
- `int high, low, results[20];`
- Declarations can be spread out, allowing space for an explanatory comment. Variables can also be initialised when they are declared, this is done by adding an equals sign and the required value after the declaration.
- `int high = 250;  /* Maximum Temperature */`
- `int low = -40;  /* Minimum Temperature */`
- `int results[20];  /* Series of temperature readings */`
- C provides a wide range of types. The most common are

  `int` An Integer
  `float` A floating point (real) number
- `char` A single byte of memory, enough to hold a character
- There are also several variants on these types.

  `short` An integer, possibly of reduced range
  `long` An integer, possibly of increased range
  `unsigned` An integer with no negative range, the spare capacity being used to increase the positive range
  `unsigned long` Like unsigned, possibly of increased range
- `double` A double precision floating point number.
- All of the integer types plus the char are called the integral types. float and double are called the real types.
- # Variable Names

- Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognises upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.
- Examples of legal variable names include
- `x` `result` `outfile` `bestyet`
- `x1` `x2` `out_file` `best_yet`
- `power` `impetus` `gamma` `hi_score`
- It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name. Most modern ones don't apply this limit though.
- The rules governing variable names also apply to the names of functions. We shall meet functions later on in the course.

Global Variables

Local variables are declared within the body of a function, and can only be used within that function. This is usually no problem, since when another function is called, all required data is passed to it as arguments. Alternatively, a variable can be declared globally so it is available to all functions. Modern programming practice recommends against the excessive use of global variables. They can lead to poor program structure, and tend to clog up the available name space.

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

External Variables

Where a global variable is declared in one file, but used by functions from another, then the variable is called an external variable in these functions, and must be declared as such. The declaration must be preceeded by the word extern. The declaration is required so the compiler can find the type of the

variable without having to search through several source files for the declaration.

Global and external variables can be of any legal type. They can be initialised, but the initialisation takes place when the program starts up, before entry to the main function.

Static Variables

Another class of local variable is the static type. A static can only be accessed from the function in which it was declared, like a local variable. The static variable is not destroyed on exit from the function, instead its value is preserved, and becomes available again when the function is next called. Static variables are declared as local variables, but the declaration is preceeded by the word static.

```
static int counter;
```

Static variables can be initialised as normal, the initialisation is performed once only, when the program starts up.

Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence.

'\n'	newline
'\t'	tab
'\\ '	backslash
'\''	single quote
'\0'	null (used automatically to terminate character strings).

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes eg "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

A character is a different type to a single character string. This is important.

We will meet strings and characters again when we deal with the input / output functions in more detail.

Arrays

An array is a collection of variables of the same type. Individual array elements are identified by an integer index. In C the index begins at zero and is always written inside square brackets.

We have already met single dimensioned arrays which are declared like this

```
int results[20];
```

Arrays can have more dimensions, in which case they might be declared as

```
int results_2d[20][5];
int results_3d[20][5][3];
```

Each index has its own set of square brackets.

Where an array is declared in the main function it will usually have details of dimensions included. It is possible to use another type called a pointer in place of an array. This means that dimensions are not fixed immediately, but space can be allocated as required. This is an advanced technique which is only required in certain specialised programs.

When passed as an argument to a function, the receiving function need not know the size of the array. So for example if we have a function which sorts a list (represented by an array) then the function will be able to sort lists of different sizes. The drawback is that the function is unable to determine what

size the list is, so this information will have to be passed as an additional argument.

As an example, here is a simple function to add up all of the integers in a single dimensioned array.

```
int add_array(int array[], int size)
{       int i;
        int total = 0;

        for(i = 0; i < size; i++)
                total += array[i];

        return(total);
}
```

Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as +, -, /.

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

- **Assignment Statement**

- The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like
- `y = (m * x) + c`
- This assignment will save the value of the expression in variable y.
- **Arithmetic operators**
- Here are the most common arithmetic operators
  - `+` Addition
  - `-` Subtraction
  - `*` Multiplication
  - `/` Division
- `%` Modulo Reduction (Remainder from integer division)
- *, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.
- Here are some arithmetic expressions used within assignment statements.
- `velocity = distance / time;`
-
- `force = mass * acceleration;`
-
- `count = count + 1;`
- C has some operators which allow abbreviation of certain types of arithmetic assignment statements.
-

Shorthand	Equivalent
`i++;` or `++i;`	`i = i + 1;`
`i--;` or `--i;`	`i = i - 1;`

- These operations are usually very efficient. They can be combined with another expression.
- `x = a * b++;` is equivalent to
```
x = a * b;
b = b + 1;
```
- Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so
- `x = --i * (a + b);` is equivalent to
```
i = i - 1;
x = i * (a + b);
```

- These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of++ and – to ensure that your programs stay readable.
- Another shorthand notation is listed below

Shorthand	Equivalent
`i += 10;`	`i = i + 10;`
`i -= 10;`	`i = i - 10;`
`i *= 10;`	`i = i * 10;`
`i /= 10;`	`i = i / 10;`

- These are simple to read and use.

Type conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size, so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.

There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;

- The variable is too small to hold the value. In this case it will be corrupted (this is bad).
- The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.

Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting which temporarily disguises a value as a different type.

eg. The function sqrt finds the square root of a double.

```
int i = 256;
int root

root = sqrt( (double) i);
```

The cast is made by putting the bracketed name of the required type just before the value. (double) in this example. The result of sqrt((double) i); is also a double, but this is automatically converted to an int on assignment to root.

Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

C notation	Meaning
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Note that == is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

```
x == y

i > 10

a + b != c
```

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement or a for or a while loop. These will be introduced in a later chapter.

Logical Connectors

These are the usual And, Or and Not operators.

Symbol	Meaning
&&	And
||	Or
!	Not

They are frequently used to combine relational operators, for example

```
x < 20 && x >= 10
```

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
        printf("Not Acceptable !!\n");
```

Summary

Three types of expression have been introduced here;

- Arithmetic expressions are simple, but watch out for subtle type conversions. The shorthand notations may save you a lot of typing.
- Comparison takes two numbers and produces a logical result. Comparisons are usually found controlling if statements or loops.
- Logical connectors allow several comparisons to be combined into a single test. Lazy evaluation can improve the efficiency of the program by reducing the amount of calculation required.

C also provides bit manipulation operators. These are too specialised for the scope of this course.

Control Statements

A program consists of a number of statements which are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.

Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

This section will discuss the use of control statements in C. We will show how they can be used to write powerful programs by;

- Repeating important sections of the program.
- Selecting between optional sections of a program.

- # The if else Statement
- This is used to decide whether to do something at a special point, or to decide between two courses of action.
- The following test decides whether a student has passed an exam with a pass mark of 45

```
if (result >= 45)
        printf("Pass\n");
else
        printf("Fail\n");
```

- It is possible to use the if part without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

- Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If

is is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

- If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{       printf("Passed\n");
        printf("Congratulations\n")
}
else
{       printf("Failed\n");
        printf("Good luck in the resits\n");
}
```

- Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
        printf("Passed: Grade A\n");
else if (result >= 60)
        printf("Passed: Grade B\n");
else if (result >= 45)
        printf("Passed: Grade C\n");
else
        printf("Failed\n");
```

- In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem

The switch Statement

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
        }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

Loops

C gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loops is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.
- **The while Loop**
- The while loop repeats a statement until the test at the top proves false.
- As an example, here is a function to return the length of a string. Remember that the string is represented as an array of characters terminated by a null character '\0'.

```
int string_length(char string[])
{       int i = 0;

        while (string[i] != '\0')
                i++;

        return(i);
}
```

- The string is passed to the function as an argument. The size of the array is not specified, the function will work for a string of any size.
- The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't null, the index is incremented and the test is repeated.

The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialisation of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

The example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{       float total = 0.0;
        int i;

        for(i = 0; i < count; i++)
                total += array[i];

        return(total / count);
}
```

The for loop ensures that the correct number of array elements are added up before calculating the average.

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialisation or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely

unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behaviour to be specified simply and quickly.

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

The goto Statement

C has a goto statement which permits unstructured jumps to be made. Its use is not recommended, so we'll not teach it here. Consult your textbook for details of its use.

Functions in C

Almost all programming languages have some equivalent of the function. You may have met them under the alternative names subroutine or procedure.

Some languages distinguish between functions which return variables and those which don't. C assumes that every function will return a value. If the programmer wants a return value, this is achieved using the return statement. If no return value is required, none should be used when calling the function.

Here is a function which raises a double to the power of an unsigned, and returns the result.

```
double power(double val, unsigned pow)
{       double ret_val = 1.0;
        unsigned i;

        for(i = 0; i < pow; i++)
                ret_val *= val;

        return(ret_val);
}
```

The function follows a simple algorithm, multiplying the value by itself pow times. A for loop is used to control the number of multiplications, and variable ret_val stores the value to be returned. Careful programming has ensured that the boundary condition is correct too. ie .

Let us examine the details of this function.

double power(double val, unsigned pow)
This line begins the function definition. It tells us the type of the return value, the name of the function, and a list of arguments used by the function. The

arguments and their types are enclosed in brackets, each pair separated by commas.

The body of the function is bounded by a set of curly brackets. Any variables declared here will be treated as local unless specifically declared as static or extern types.

return(ret_val);
On reaching a return statement, control of the program returns to the calling function. The bracketed value is the value which is returned from the function. If the final closing curly bracket is reached before any return value, then the function will return automatically, any return value will then be meaningless.

The example function can be called by a line in another function which looks like this

```
result = power(val, pow);
```

This calls the function power assigning the return value to variable result.

Here is an example of a function which does not return a value.

```
void error_line(int line)
{      fprintf(stderr, "Error in input data: line %d\n", line);
}
```

The definition uses type void which is optional. It shows that no return value is used. Otherwise the function is much the same as the previous example, except that there is no return statement. Some void type functions might use return, but only to force an early exit from the function, and not to return any value. This is rather like using break to jump out of a loop.

This function also demonstrates a new feature.

fprintf(stderr, "Error in input data: line %d\n", line);
This is a variant on the printf statement, fprintf sends its output into a file. In this case, the file is stderr. stderr is a special UNIX file which serves as the channel for error messages. It is usually connected to the console of the computer system, so this is a good way to display error messages from your

programs. Messages sent to stderr will appear on screen even if the normal output of the program has been redirected to a file or a printer.

The function would be called as follows

```
error_line(line_number);
```

Scope of Function Variables

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

Local variables are declared within a function. They are created anew each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

Static variables are slightly different, they don't die on return from the function. Instead their last value is retained, and it becomes available when the function is called again.

Global variables don't die on return from a function. Their value is retained, and is available to any other function which accesses them.

Modifying Function Arguments

Some functions work by modifying the values of their arguments. This may be done to pass more than one value back to the calling routine, or because the return value is already being used in some way. C requires special arrangements for arguments whose values will be changed.

You can treat the arguments of a function as variables, however direct manipulation of these arguments won't change the values of the arguments in the calling function. The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.

There is a way to change the values of variables declared outside the function. It is done by passing the addresses of variables to the function. These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer. It is what lies at the address which the pointer references which interests us.

To get back to our original function, we pass it the address of a variable whose value we wish to change. The function must now be written to use the value at that address (or at the end of the pointer). On return from the function, the desired value will have changed. We manipulate the actual value using a copy of the pointer.

Pointers in C

Pointers are not exclusive to functions, but this seems a good place to introduce the pointer type.

Imagine that we have an int called i. Its address could be represented by the symbol &i. If the pointer is to be stored as a variable, it should be stored like this.

```
int *pi = &i;
```

int * is the notation for a pointer to an int. & is the operator which returns the address of its argument. When it is used, as in &i we say it is referencing i.

The opposite operator, which gives the value at the end of the pointer is *. An example of use, known as de-referencing pi, would be

```
i = *pi;
```

Take care not to confuse the many uses of the * sign; Multiplication, pointer declaration and pointer de-referencing.

This is a very confusing subject, so let us illustrate it with an example. The following function fiddle takes two arguments, x is an int while y is a pointer to int. It changes both values.

```
fiddle(int x, int *y)
{   printf(" Starting fiddle: x = %d, y = %d\n", x, *y);
    x ++;
    (*y)++;
    printf("Finishing fiddle: x = %d, y = %d\n", x, *y);
}
```

since y is a pointer, we must de-reference it before incrementing its value.

A very simple program to call this function might be as follows.

```
main()
{   int i = 0;
    int j = 0;

    printf(" Starting main  : i = %d, j = %d\n", i, j);
    printf("Calling fiddle now\n");.
    fiddle(i, &j);
    printf("Returned from fiddle\n");
    printf("Finishing main  : i = %d, j = %d\n", i, j);
}
```

Note here how a pointer to int is created using the & operator within the call fiddle(i, &j);.

The result of running the program will look like this.

```
 Starting main  : i =   0 ,j =   0
Calling fiddle now
 Starting fiddle: x =   0, y =   0
Finishing fiddle: x =   1, y =   1
Returned from fiddle
Finishing main  : i =   0, j =   1
```

After the return from fiddle the value of i is unchanged while j, which was passed as a pointer, has changed.

To summarise, if you wish to use arguments to modify the value of variables from a function, these arguments must be passed as pointers, and de-referenced within the function.

Where the value of an argument isn't modified, the value can be passed without any worries about pointers.

Arrays and Pointers

To fully understand the workings of C you must know that pointers and arrays are related.

An array is actually a pointer to the 0th element of the array. Dereferencing the array name will give the 0th element. This gives us a range of equivalent notations for array access. In the following examples, arr is an array.

Array Access	Pointer Equivalent
`arr[0]`	`*arr`
`arr[2]`	`*(arr + 2)`
`arr[n]`	`*(arr + n)`

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the values in the array, but not the array itself, so statements like `arr ++` are illegal, but `arr[n] ++` is legal.

Since an array is like a pointer, we can pass an array to a function, and modify elements of that array without having to worry about referencing and de-referencing. Since the array is implemented as a hidden pointer, all the difficult stuff gets done automatically.

A function which expects to be passed an array can declare that parameter in one of two ways.

```
int arr[];  or int *arr;
```

Either of these definitions is independent of the size of the array being passed. This is met most frequently in the case of character strings, which are implemented as an array of type char. This could be declared as char string[]; but is most frequently written as char *string; In the same way, the argument vector argv is an array of strings which can be supplied to function main. It can be declared as one of the following.

```
char **argv;  or  char *argv[];
```

Don't panic if you find pointers confusing. While you will inevitably meet pointers in the form of strings, or as variable arguments for functions, they need not be used in most other simple types of programs.

Recursive Functions

A recursive function is one which calls itself. This is another complicated idea which you are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{       switch(num) {
        case 0:
                return(0);
                break;
        case 1:
                return(1);
                break;
        default:  /* Including recursive calls */
                return(fib(num - 1) + fib(num - 2));
                break;
        }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
        if(pow == 0)  /* pow(x, 0) returns 1 */
                return(1.0);
        else
                return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly, otherwise the function calls itself,

passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the fibonacci function of a moderate size number.

Input Value	Number of times `fib` is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analysing them and storing results in variables. These variants of the basic input and output commands are discussed in the next section

---

- # The Standard Input Output File
- UNIX supplies a standard package for performing input and output to files or the terminal. This contains most of the functions which will be introduced in this section, along with definitions of the datatypes required to use them. To use these facilities, your program must include these definitions by adding the line This is done by adding the line
- ```
  #include <stdio.h>
  ```
- near the start of the program file.
- If you do not do this, the compiler may complain about undefined functions or datatypes.

Character Input / Output

This is the lowest level of input and output. It provides very precise control, but is usually too fiddly to be useful. Most computers perform buffering of input and output. This means that they'll not start reading any input until the return key is pressed, and they'll not print characters on the terminal until there is a whole line to be printed.

  ```

getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>

main()
{   int ch, i = 0;

    while((ch = getchar()) != EOF)
        i ++;

    printf("%d\n", i);
}
```

putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function toupper from the character conversion library ctype.h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

main()
{   int ch;

    while((ch = getchar()) != EOF)
        putchar(toupper(ch));
}
```

Formatted Input / Output

We have met these functions earlier in the course. They are closest to the facilities offered by Pascal or Fortran, and usually the easiest to use for input and output. The versions offered under C are a little more detailed, offering precise control of layout.

- printf
- scanf

printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

There are several more types available. For full details type

```
man printf
```

on your UNIX system.

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field.

scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceeded by the &

sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceeded by the percentage sign in a similar way to their printf equivalents.

Type man scanf for details of all options on your system.

Whole Lines of Input and Output

Where we are not too interested in the format of our data, or perhaps we cannot predict its format in advance, we can read and write whole lines as character strings. This approach allows us to read in a line of input, and then use various string handling functions to analyse it at our leisure.

- gets
- puts

gets

gets reads a whole line of input into a string until a newline or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in stdio.h is returned.

puts

puts writes a string to the output, and follows it with a newline character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>

main()
{       char line[256]; /* Define string sufficiently large to
                           store a line of input */

        while(gets(line) != NULL)   /* Read line        */
        {       puts(line);         /* Print line       */
```

```
                printf("\n");       /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together. So can getchar, scanf and gets.