C Programming

Introduction (Creating, Compiling and Running Your Program)

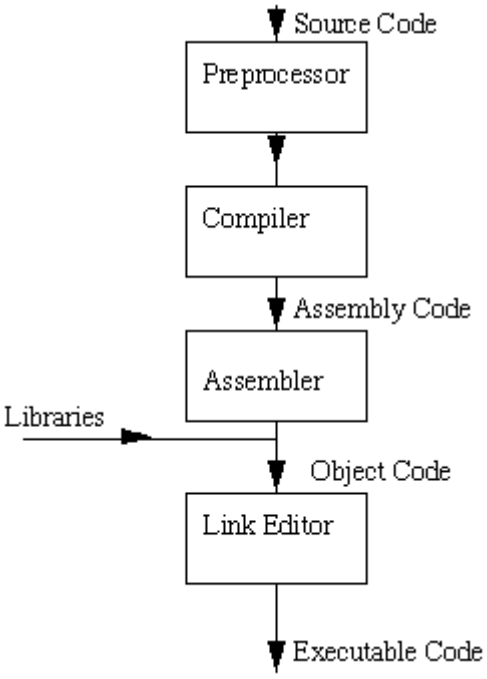
Creating the program

Create a file containing the complete program, such as the above example. You can use any ordinary editor with which you are familiar to create the file. For example, one such editor is *textedit* available on most UNIX systems.

The filename must by convention end ``.c" (full stop, lower case c), *e.g. myprog.c* or *progtest.c*. The contents must obey C syntax. For example, they might be as in the above example, starting with the line `/* Sample ....` (or a blank line preceding it) and ending with the line `} /* end of program */` (or a blank line following it).

Compilation

We will briefly highlight key features of the C Compilation model:



There are many C compilers around. The `cc` being the default Sun compiler. The GNU C compiler `gcc` is popular and available for many platforms. PC users may also be familiar with the Borland `bcc` compiler.

The Preprocessor

We will study this part of the compilation process in greater detail later. However we need some basic information for some C programs.

The Preprocessor accepts source code as input and is responsible for

- removing comments
- interpreting special *preprocessor directives* denoted by `#`.

For example

- `#include` -- includes contents of a named file. Files usually called *header* files. *e.g*
  - `#include <math.h>` -- standard library maths file.
  - `#include <stdio.h>` -- standard library I/O file
- `#define` -- defines a symbolic name or constant. Macro substitution.
  - `#define MAX_ARRAY_SIZE 100`

C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler

The assembler creates object code. On a UNIX system you may see files with a `.o` suffix (`.OBJ` on MSDOS) to indicate object code files.

Link Editor

If a source file references library functions or functions defined in other source files the *link editor* combines these functions (with `main()`) to create an executable file. External Variable references resolved here also. *More on this later*.

Some Useful Compiler Options

Now that we have a basic understanding of the compilation model we can now introduce some useful and sometimes essential common compiler options. Again see the online `man` pages and Appendix ❑ for further information and additional options.

`-c`

Suppress the linking process and produce a `.o` file for each source file listed. Several can be subsequently linked by the `cc` command, for example:

```
cc file1.o file2.o ..... -o executable
```

`-llibrary`

Link with object libraries. This option must follow the source file arguments. The object libraries are archived and can be standard, third party or user created libraries (We discuss this topic briefly below and also in detail later. Probably the most commonly used library is the math library (`math.h`). You must link in this library explicitly if you wish to use the maths functions (**note** do note forget to `#include <math.h>` header file), for example:

```
cc calc.c -o calc -lm
```

Many other libraries are linked in this fashion (see below)

`-Ldirectory`

Add directory to the list of directories containing object-library routines. The linker always looks for standard and other system libraries in `/lib` and `/usr/lib`. If you want to link in libraries that you have created or installed yourself (unless you have certain privileges and get the libraries installed in `/usr/lib`) you **will** have to specify where you files are stored, for example:

```
cc prog.c -L/home/myname/mylibs mylib.a
```

Running the program

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case *program* (or *a.out*)

This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.

If so, you must return to edit your program source, and recompile it, and run it again.

Exercises

1. Enter, compile and run the following program:

```
main()
{ int i;
   printf("\t Number \t\t Square of Number\n\n");
   for (i=0; i<=25;++i)
```

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

        }
```

2. The following program uses the math library. Enter compile and run it correctly.

```
#include <math.h>

        main()

        { int i;

           printf("\t Number \t\t Square Root of Number\n\n");

           for (i=0; i<=360; ++i)
            printf("\t %d \t\t\t %d \n",i, sqrt((double) i));

        }
```

C Basics

Before we embark on a brief tour of C's basic syntax and structure we offer a brief history of C and consider the characteristics of the C language.

In the remainder of the Chapter we will look at the basic aspects of C programs such as C program structure, the declaration of variables, data types and operators. We will assume knowledge of a high level language, such as PASCAL.

It is our intention to provide a quick guide through similar C principles to most high level languages. Here the syntax may be slightly different but the concepts exactly the same.

C does have a few surprises:

- Many High level languages, like PASCAL, are highly disciplined and structured.
- **However beware** -- C is much more flexible and free-wheeling. This freedom gives C much more **power** that experienced users can employ. The above example below (`mystery.c`) illustrates how bad things could really get.

History of C

The *milestones* in C's development as a language are listed below:

- UNIX developed c. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language ``B" a second attempt. c. 1970.
- A totally new language ``C" a successor to ``B". c. 1971
- By 1973 UNIX OS almost totally written in ``C".

Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

As an extreme example the following C code (`mystery.c`) is actually *legal* C code.

Clearly nobody ever writes code like or at least should never. This piece of code actually one an international Obfuscated C Code Contest http://reality.sgi.com/csp/iocc The standard for C programs was originally the features set by Brian Kernighan. In order to make the language more internationally acceptable, an international standard was developed, ANSI C (American National Standards Institute).

C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

We must have a `main()` function.

A function has the form:

```
type function_name (parameters)
                {
                                local variables

                                C Statements

                 }
```

If the type definition is omitted C assumes that function returns an **integer** type. **NOTE:** This can be a source of problems in a program.

So returning to our first C program:

```
/* Sample program */

             main()
                                {

                                                    \
                                printf( ``I Like C   'n'' );
                                exit ( 0 );
```

Variables

C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	-32768	$+32767$
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{+38}$	$+3.2 \times 10^{+38}$
double	8	$-1.7 \times 10^{+308}$	$+1.7 \times 10^{+308}$

The Pascal Equivalents are:

C type	Pascal equivalent
char	char
unsigned char	—
short int	integer
unsigned short int	—
long int	longint
float	real
double	extended

On UNIX systems all `int`s are `long int`s unless specified as `short int` explicitly.

NOTE: There is **NO** Boolean type in C -- you should use `char, int` or (better) `unsigned char`.

`Unsigned` can be used with all `char` and `int` types.

To declare a variable in C, do:

`var_type` *list variables*;

```
e.g. int i,j,k;
                float x,y,z;
                char ch;
```

Defining Global Variables

Global variables are defined above `main()` in the following way:-

```
          short number,sum;
                int bignumber,bigsum;
                char letter;

                main()
                                {

                                }
```

It is also possible to pre-initialise global variables using the `= operator for assignment.`

NOTE: `The = operator is the same as := is Pascal.`

`For example:-`

```
          float sum=0.0;
```

```
int bigsum=0;
char letter=`A';

main()
                {

                }
```

This is the same as:-

```
float sum;
      int bigsum;
      char letter;

      main()
                {

                sum=0.0;
                bigsum=0;
                letter=`A';

                }
```

...but is more efficient.

C also allows multiple assignment statements using =, for example:

```
a=b=c=d=3;
```

...which is the same as, but more efficient than:

```
a=3;
     b=3;
     c=3;
     d=3;
```

This kind of assignment is only possible if all the variable types in the statement are the same.

You can define your own types use typedef. This will have greater relevance later in the course when we learn how to create more complex data structures.

As an example of a simple use let us consider how we may define two new types real and letter. These new types can then be used in the same way as the pre-defined C types:

```
typedef real float;
      typedef letter char;
```

```
Variables declared:
                real sum=0.0;
                letter nextletter;
```

Printing Out and Inputting Variables

C uses formatted output. The `printf` function has a special formatting character (%) -- a character following this defines a certain format for a variable:

```
   %c -- characters
               %d -- integers
               %f -- floats
```

e.g. `printf(``%c %d %f'',ch,i,x);`

NOTE: Format statement enclosed in ``...", variables follow after. Make sure order of format and variable data types match up.

`scanf()` is the function for inputting values to a data structure: Its format is similar to `printf`:

i.e. `scanf(``%c %d %f'',&ch,&i,&x);`

NOTE: & before variables. Please accept this for now and **remember** to include it. It is to do with pointers which we will meet later.

Constants

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The `const` keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

Note:

- You can declare the `const` before or after the type. Choose one an stick to it.
- It is usual to initialise a `const` with a value as it cannot get a value *any other way*.

The preprocessor `#define` is another more flexible (see Preprocessor Chapters) method to define *constants* in a program.

You frequently see const declaration in function parameters. This says simply that the function is **not** going to change the value of the parameter.

The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completenes of this section it is is included here:

```
void strcpy(char *buffer, char const *string)
```

The second argiment `string` is a C string that will not be altered by the string copying standard library function.

Arithmetic Operations

As well as the standard arithmetic operators (`+ - * /`) found in most languages, C provides some more operators. There are some notable differences with other languages, such as Pascal.

Assignment is = *i.e. i* = 4; `ch` = `y';

Increment ++, Decrement - which are more efficient than their long hand equivalents, for example:- `x++` is faster than `x=x+1`.

The `++` and `-` operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

In the example below, `++z` is pre-fixed and the `w-` is post-fixed:

```
int x,y,w;

      main()
                        {

                        x=((++z)-(w-)) % 100;

                        }
```

This would be equivalent to:

```
int x,y,w;

      main()
                        {

                       z++;
                       x=(z-w) % 100;
```

```
w-;
 }
```

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to: *x* = 3 / 2 is 1 even if *x* is declared a float!!

RULE: If both arguments of / are integer then do integer division.

So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 / 2.0.

There is also a convenient **shorthand** way to express computations in C.

It is very common to have expressions like: *i* = *i* + 3 or *x* = *x*(*y* + 2)

This can written in C (generally) in a *shorthand* form like this:

$$expr_1\ op\ =\ expr_2$$

which is equivalent to (but more efficient than):

$$expr_1\ =\ expr_1\ op\ expr_2$$

So we can rewrite *i* = *i* + 3 as *i* += 3

and *x* = *x*(*y* + 2) as *x* *= *y* + 2.

NOTE: that *x* *= *y* + 2 means *x* = *x*(*y* + 2) and **NOT** *x* = *x***y* + 2.

Comparison Operators

To test for equality is ==

A warning: Beware of using ``=" instead of ``==", such as writing accidentally

```
if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Logical Operators

Logical operators are usually used with conditional statements which we shall meet in the next Chapter.

The two basic logical operators are:

&& for logical AND, || for logical OR.

Beware & and | have a different meaning for bitwise AND and OR.

Order of Precedence

It is necessary to be careful of the meaning of such expressions as `a + b * c`

We may want the effect as either

```
(a + b) * c
```

or

```
a + (b * c)
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
a - b - c
```

is evaluated as

```
( a - b ) - c
```

as you would expect.

From high priority to low priority the order for all C operators (we have not met all of them yet) is:

```
( )  [  ]  -> .
           !  ~ - * & sizeof cast ++ -
                 (these are right->left)
           * / %
           + -
           < <= >= >
           == !=
```

```
&
^             |
&&
||
?:        (right->left)
= += -= (right->left)
,   (comma)
```

Thus

```
a < 10 && 2 * b < c
```

is interpreted as

```
( a < 10 ) && ( ( 2 * b ) < c )
```

and

```
a =

            b =
                        spokes / spokes_per_wheel
                        + spares;
```

as

```
a =

            ( b =
                        ( spokes / spokes_per_wheel )
                        + spares
            );
```

Exercises

Write C programs to perform the following tasks.

1. Input two numbers and work out their sum, average and sum of the squares of the numbers.

2. Input and output your name, address and age to an appropriate structure.

3. Write a program that works out the largest and smallest values from a set of 10 inputted numbers.

3. Write a program to read a "float" representing a number of degrees Celsius, and print as a "float" the equivalent temperature in degrees Fahrenheit. Print your results in a form such as

100.0 degrees Celsius converts to 212.0 degrees Fahrenheit.

4. Write a program to print several lines (such as your name and address). You may use either several printf instructions, each with a newline character in it, or one printf with several newlines in the string.

5. Write a program to read a positive integer at least equal to 3, and print out all possible permutations of three positive integers less or equal to than this value.

Conditionals

This Chapter deals with the various methods that C can control the *flow* of logic in a program. Apart from slight syntactic variation they are similar to other languages.

As we have seen following logical operations exist in C:

==, !=, || , &&.

One other operator is the unitary - it takes only one argument - *not* **!**.

These operators are used in conjunction with the following statements.

The `if` statement

The `if` statement has the same function as other languages. It has three basic forms:

```
if  (expression)
                              statement
```

...or:

```
if  (expression)
                              statement1
                else
                              statement2
```

...or:

```
if  (expression)
                              statement1
                else if (expression)
                              statement2
                else
                              statement3
```

For example:-

```
int x,y,w;

              main()
                                {

                                if (x>0)
                                                {
                                                z=w;
                                                ........
                                                }
                                else
                                                {
                                                z=y;
                                                ........
                                                }

                                }
```

The ? operator

The ? (*ternary condition*) operator is a more efficient form for expressing simple `if` statements. It has the following form:

expression$_1$? *expression*$_2$: *expression*$_3$

It simply states:

`if` *expression*$_1$ `then` *expression*$_2$ `else` *expression*$_3$

For example to assign the maximum of `a` and `b` to `z`:

```
z = (a>b) ? a : b;
```

which is the same as:

```
if (a>b)
                                z = a;
              else
                                z=b;
```

The `switch` statement

The C `switch` is similar to Pascal's `case` statement and it allows multiple choice of a selection of items at one level of a conditional where it is a far neater way of writing multiple `if` statements:

```
switch (expression) {
                            case item1:
                                          statement1;
                                          break;
                            case item2:
                                          statement2;
                                          break;
                                          :
                                          statementn;
                                          break;
                            default:
                                          statement;
                                          break;
                           }
```

In each case the value of $item_i$ must be a constant, variables are not allowed.

The `break` is needed if you want to terminate the `switch` after execution of one choice. Otherwise the next case would get evaluated. **Note:** This is unlike most other languages.

We can also have **null** statements by just including a ; or let the switch statement *fall through* by omitting any statements (see *e.g.* below).

The `default` case is optional and catches any other cases.

For example:-

```
          switch (letter)
                           {
                            case `A':
                            case `E':
                            case `I':
                            case `O':
                            case `U':
                                          numberofvowels++;
                                          break;

                            case ` ':
                                          numberofspaces++;
                                          break;

                            default:
                                          numberofconstants++;
```

```
                                        break;
                                }
```

In the above example if the value of `letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented.`

`If the value of letter is ` ' then numberofspaces is incremented.`

`If none of these is true then the default condition is executed, that is numberofconstants is incremented.`

Exercises

1. Write a program to read two characters, and print their value when interpreted as a 2-digit hexadecimal number. Accept upper case letters for values from 10 to 15.

2. Read an integer value. Assume it is the number of a month of the year; print out the name of that month.

3. Given as input three integers representing a date as day, month, year, print out the number day, month and year for the following day's date.

Typical input: 28 2 1992 Typical output: Date following 28:02:1992 is 29:02:1992

4. Write a program which reads two integer values. If the first is less than the second, print the message up. If the second is less than the first, print the message down If the numbers are equal, print the message equal If there is an error reading the data, print a message containing the word Error and perform exit(0);

Looping and Iteration

This chapter will look at C's mechanisms for controlling looping and iteration. Even though some of these mechanisms may look familiar and indeed will operate in standard fashion most of the time. **NOTE:** some non-standard features are available.

The `for` statement

The C `for` statement has the following form:

```
for  (expression1; 2; expression3)
                              statement;
                              or {block of statements}
```

expression$_1$ initialises; *expression*$_2$ is the terminate test; *expression*$_3$ is the modifier (which may be more than just simple increment);

NOTE: C basically treats `for` statements as `while` type loops

For example:

```
int x;

               main()
                              {
                             for (x=3;x>0;x-)
                                                {
                                                                \
                                                printf("x=%d n",x);
                                                }
                              }
```

...outputs:

```
x=3
              x=2
              x=1
```

...to the screen

All the following are legal for statements in C. The practical application of such statements is not important here, we are just trying to illustrate peculiar features of C for that may be useful:-

```
for (x=0;((x>3) && (x<9)); x++)

               for (x=0,y=4;((x>3) && (y<9)); x++,y+=2)

               for (x=0,y=4,z=4000;z; z/=10)
```

The second example shows that multiple expressions can be separated a ,.

In the third example the loop will continue to iterate until z becomes 0;

The `while` statement

The `while` statement is similar to those used in other languages although more can be done with the `expression` statement -- a standard feature of C.

The `while` has the form:

```
while (expression)
                              statement
```

For example:

```
int x=3;

              main()
                            { while (x>0)
                                                    \
                                          { printf("x=%d n",x);
                                                    x-;
                                          }
                            }
```

...outputs:

```
x=3
             x=2
             x=1
...to the screen.
```

Because the while loop can accept expressions, not just conditions, the following are all legal:-

```
while (x-);
             while (x=x+1);
             while (x+=5);
```

Using this type of expression, only when the result of x-, x=x+1, or x+=5, evaluates to 0 will the while condition fail and the loop be exited.

We can go further still and perform complete operations within the while *expression*:

```
while (i++ < 10);

             while ( (ch = getchar()) != `q')
               putchar(ch);
```

The first example counts i up to 10.

The second example uses C standard library functions getchar() - reads a character from the keyboard - and putchar() - writes a given char to screen. The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read. **NOTE:** This type of operation is used a lot in C and not just with character reading!! (See Exercises).

The `do-while` statement

C's `do-while` statement has the form:

```
do
                                statement;
                                while (expression);
```

It is similar to PASCAL's `repeat` ... `until` except `do while` *expression* is true.

For example:

```
int x=3;

                main()
                                { do {

                                                printf("x=%d \n",x-);
                                                }
                                while (x>0);
                                }
```

..outputs:-

```
x=3
                x=2
                x=1
```

NOTE: The postfix x- operator which uses the current value of x while printing and *then* decrements x.

`break` and `continue`

C provides two commands to control how we loop:

- `break` -- exit form loop or switch.
- `continue` -- skip 1 iteration of loop.

Consider the following example where we read in integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

```
while (scanf( ``%d'', &value ) == 1 && value != 0) {
```

```
                                if (value < 0) {
                                                                        \
                                        printf(``Illegal value
n'');
                                        break;
                                        /* Abandon the loop */
                                }

                                if (value > 100) {
                                                                        \
                                        printf(``Invalid value
n'');
                                        continue;
                                        /* Skip to start loop again
*/
                                }

                                /* Process the value read */
                                  /* guaranteed between 1 and 100 */
                                     ....;

                                ....;
                        } /* end while value != 0 */
```

Exercises

1. Write a program to read in 10 numbers and compute the average, maximum and minimum values.

2. Write a program to read in numbers until the number -999 is encountered. The sum of all number read until this point should be printed out.

3. Write a program which will read an integer value for a base, then read a positive integer written to that base and print its value.

Read the second integer a character at a time; skip over any leading non-valid (i.e. not a digit between zero and ``base-1'') characters, then read valid characters until an invalid one is encountered.

```
          Input       Output
        ==========    ======
        10    1234     1234
         8      77       63   (the value of 77 in base 8, octal)
         2    1111       15   (the value of 1111 in base 2, binary)
```

The base will be less than or equal to 10.

4. Read in three values representing respectively

a capital sum (integer number of pence),

a rate of interest in percent (float),

and a number of years (integer).

Compute the values of the capital sum with compound interest added over the given period of years. Each year's interest is calculated as

interest = capital * interest_rate / 100;

and is added to the capital sum by

capital += interest;

Print out money values as pounds (pence / 100.0) accurate to two decimal places.

Print out a floating value for the value with compound interest for each year up to the end of the period.

Print output year by year in a form such as:

```
Original sum 30000.00 at  12.5 percent for 20 years

Year Interest  Sum
----+-------+--------
  1  3750.00 33750.00
  2  4218.75 37968.75
  3  4746.09 42714.84
  4  5339.35 48054.19
  5  6006.77 54060.96
  6  6757.62 60818.58
  7  7602.32 68420.90
  8  8552.61 76973.51
  9  9621.68 86595.19
 10 10824.39 97419.58
```

5. Read a positive integer value, and compute the following sequence: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value. Finally print out how many of these operations you performed.

Typical output might be:

```
 Inital value is 9
 Next value is  28
 Next value is  14
 Next value is   7
 Next value is  22
 Next value is  11
 Next value is  34
 Next value is  17
 Next value is  52
 Next value is  26
 Next value is  13
 Next value is  40
```

```
 Next value is  20
 Next value is  10
 Next value is   5
 Next value is  16
 Next value is   8
 Next value is   4
 Next value is   2
 Final value 1, number of steps 19
```

If the input value is less than 1, print a message containing the word

```
Error
```

and perform an

```
exit( 0 );
```

6. Write a program to count the vowels and letters in free text given as standard input. Read text a character at a time until you encounter end-of-data.

Then print out the number of occurrences of each of the vowels a, e, i, o and u in the text, the total number of letters, and each of the vowels as an integer percentage of the letter total.

Suggested output format is:

```
Numbers of characters:
a   3 ; e   2 ; i   0 ; o   1 ; u   0 ; rest  17
Percentages of total:
a  13%; e   8%; i   0%; o   4%; u   0%; rest  73%
```

Read characters to end of data using a construct such as

```
char ch;
while(
    ( ch = getchar() ) >= 0
) {
    /* ch is the next character */    ....
}
```

to read characters one at a time using `getchar()` until a negative value is returned.

Single and Multi-dimensional Arrays

Let us first look at how we define arrays in C:

```
int listofnumbers[50];
```

BEWARE: In C Array subscripts start at **0** and end one less than the array size. For example, in the above case valid subscripts range from 0 to 49. This is a **BIG** difference between C and other languages and does require a bit of practice to get in *the right frame of mind*.

Elements can be accessed in the following ways:-

```
thirdnumber=listofnumbers[2];
             listofnumbers[5]=100;
```

```
Multi-dimensional arrays can be defined as follows:
```

```
  int tableofnumbers[50][50];
```

for two dimensions.

For further dimensions simply add more [ ]:

```
  int bigD[50][50][40][30]......[50];
```

Elements can be accessed in the following ways:

```
  anumber=tableofnumbers[2][3];
               tableofnumbers[25][16]=100;
```

Strings

In C Strings are defined as arrays of characters. For example, the following defines a string of 50 characters:

```
  char name[50];
```

C has no string handling facilities built in and so the following are all illegal:

```
  char firstname[50],lastname[50],fullname[100];

               firstname= "Arnold"; /* Illegal */
               lastname= "Schwarznegger"; /* Illegal */
               fullname= "Mr"+firstname
                             +lastname; /* Illegal */
```

However, there is a special library of string handling routines which we will come across later.

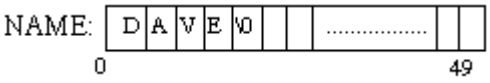
To print a string we use printf with a special **%s** control character:

```
   printf(``%s'',name);
```

NOTE: We just need to give the name of the string.

In order to allow variable length strings the \0 character is used to indicate the end of a string.

So we if we have a string, char NAME[50]; and we store the ``DAVE'' in it its contents will look like:



Exercises

1. Write a C program to read through an array of any type. Write a C program to scan through this array to find a particular value.

2. Read ordinary text a character at a time from the program's standard input, and print it with each line reversed from left to right. Read until you encounter end-of-data (see below).

You may wish to test the program by typing

```
prog5rev | prog5rev
```

to see if an exact copy of the original input is recreated.

To read characters to end of data, use a loop such as either

```
char ch;
while( ch = getchar(), ch >= 0 ) /* ch < 0 indicates end-of-data
*/
```

or

```
char ch;
while( scanf( "%c", &ch ) == 1 ) /* one character read */
```

3. Write a program to read English text to end-of-data (type control-D to indicate end of data at a terminal, see below for detecting it), and print a count of word lengths, i.e. the total number of words of length 1 which occurred, the number of length 2, and so on.

Define a word to be a sequence of alphabetic characters. You should allow for word lengths up to 25 letters.

Typical output should be like this:

```
      length 1 : 10 occurrences
             length 2 : 19 occurrences
length 3 : 127 occurrences
       length 4 : 0 occurrences
      length 5 : 18 occurrences
             ....
```

To read characters to end of data see above question.

Functions

C provides functions which are again similar most languages. One difference is that C regards `main()` as function. Also unlike some languages, such as Pascal, C does not have *procedures* -- it uses functions to service both requirements.

Let us remind ourselves of the form of a function:

```
  returntype fn_name(1, parameterdef2,...)


                                {
                                 localvariables
                                 functioncode
                                 }
```

Let us look at an example to find the average of two integers:

```
  float findaverage(float a, float b)
                                { float average;
>                                average=(a+b)/2;
                                                return(average);
                                }
```

We would *call* the function as follows:

```
                 main()
                                {               float a=5,b=15,result;
                                                result=findaverage(a,b);
                                                printf("average=%f
n",result);
                                }
```

Note: The return statement passes the result back to the main program.

`void` functions

The `void` function provide a way of emulating PASCAL type procedures.

If you do not want to return a value you must use the return type `void` and miss out the `return` statement:

```
  void squares()
                                { int loop;
                                                for (loop=1;loop<10;loop++);
                                                                printf("%d
n",loop*loop);
```

```
                        }

        main()

                        {               squares();
                        }
```

NOTE: We must have () even for no parameters unlike some languages.

Functions and Arrays

Single dimensional arrays can be passed to functions as follows:-

```
  float findaverage(int size,float list[])

                        { int i;
                                        float sum=0.0;

                                        for (i=0;i<size;i++)

sum+=list[i];
                                        return(sum/size);
                        }
```

Here the declaration `float list[]` tells C that list is an array of float. **Note** we do not specify the dimension of the array when it is a *parameter* of a function.

Multi-dimensional arrays can be passed to functions as follows:

Here `float table[][5]` tells C that table is an array of dimension N × 5 of float. **Note** we must specify the second (and subsequent) dimension of the array BUT not the first dimension.

Function Prototyping

Before you use a function C must have *knowledge* about the type it returns and the parameter types the function expects.

The ANSI standard of C introduced a new (better) way of doing this than previous versions of C. (Note: All new versions of C now adhere to the ANSI standard.)

The importance of prototyping is twofold.

- It makes for more structured and therefore easier to read code.

- It allows the C compiler to check the *syntax* of function calls.

How this is done depends on the scope of the function. Basically if a functions has been defined before it is used (called) then you are ok to merely use the function.

If NOT then you must *declare* the function. The declaration simply states the type the function returns and the type of parameters used by the function.

It is usual (and therefore **good**) practice to prototype all functions at the start of the program, although this is not strictly necessary.

To *declare* a function prototype simply state the type the function returns, the function name and in brackets list the type of parameters in the order they appear in the function definition.

e.g.

```
int strlen(char []);
```

This states that a function called `strlen` returns an integer value and accepts a single string as a parameter.

NOTE: Functions can be prototyped and variables defined on the same line of code. This used to be more popular in pre-ANSI C days since functions are usually prototyped separately at the start of the program. This is still perfectly legal though: order they appear in the function definition.

e.g.

```
int length, strlen(char []);
```

Here `length` is a variable, `strlen` the function as before.

Exercises

1. Write a function ``replace" which takes a pointer to a string as a parameter, which replaces all spaces in that string by minus signs, and delivers the number of spaces it replaced.

Thus

```
        char *cat = "The cat sat";
        n = replace( cat );
```

should set

```
        cat to "The-cat-sat"
```

and

```
        n to 2.
```

2. Read two integers, representing a rate of pay (pence per hour) and a number of hours. Print out the total pay, with hours up to 40 being paid at basic rate, from 40 to 60 at rate-and-a-half, above 60 at double-rate. Print the pay as pounds to two decimal places.

Terminate the loop when a zero rate is encountered. At the end of the loop, print out the total pay.

The code for computing the pay from the rate and hours is to be written as a function.

The recommended output format is something like:

```
Pay at 200 pence/hr for 38 hours is 76.00 pounds
Pay at 220 pence/hr for 48 hours is 114.40 pounds
Pay at 240 pence/hr for 68 hours is 206.40 pounds
Pay at 260 pence/hr for 48 hours is 135.20 pounds
Pay at 280 pence/hr for 68 hours is 240.80 pounds
Pay at 300 pence/hr for 48 hours is 156.00 pounds
Total pay is 928.80 pounds
```

The ``program features" checks that explicit values such as 40 and 60 appear only once, as a `#define` or initialised variable value. This represents good programming practice.

Pointers

Pointer are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

C uses *pointers* a lot. **Why?**:

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

C uses pointers explicitly with:

- Arrays,
- Structures,
- Functions.

NOTE: Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different DIFFERENT from other languages.

What is a Pointer?

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The *unary* or *monadic* operator **&** gives the ``address of a variable".

The *indirection* or dereference operator ***** gives the ``contents of an object *pointed to* by a pointer".

To declare a pointer to a variable do:

```
int *pointer;
```

NOTE: We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

Consider the effect of the following code:

```
   int x = 1, y = 2;
                int *ip;

                ip = &x;



y = *ip;




x = ip;



                *ip = 3;
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer work. Consider Fig. 9.1. Assume for the sake of this discussion that variable `x resides at memory location 100, y at 200 and ip at 1000.` **`Note`** `A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is` *`new`*`.`

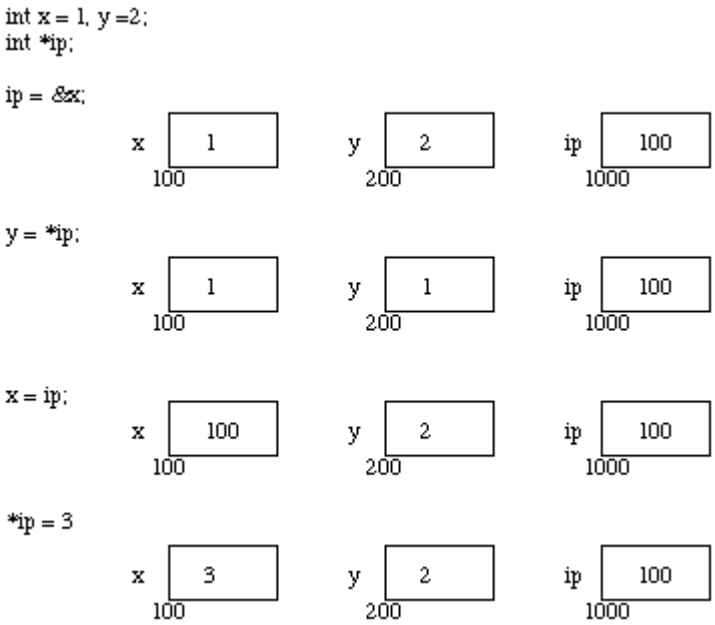


Fig. 9.1 Pointer, Variables and Memory Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a *pointer to an integer* and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the *contents of* ip. In this example ip currently *points* to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1.

We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly **legal** (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 100.

Finally we can assign a value to the contents of a pointer (*ip).

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

```
int *ip;

            *ip = 100;
```

will generate an error (program crash!!).

The correct use is:

```
int *ip;
```

```
                int x;

                ip = &x;
                *ip = 100;
```

We can do integer arithmetic on a pointer:

```
   float *flp, *flq;

                *flp = *flp + 10;

                ++*flp;

                (*flp)++;

                flq = flp;
```

NOTE: A pointer to any variable type is an address in memory -- which is an integer address. A pointer is definitely NOT an integer.

The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one ``block'' memory.

So for a character pointer ++ch_ptr adds 1 byte to the address.

For an integer or float ++ip or ++flp adds 4 bytes to the address.

Consider a float variable (fl) and a pointer to a float (flp) as shown in Fig. 9.2.

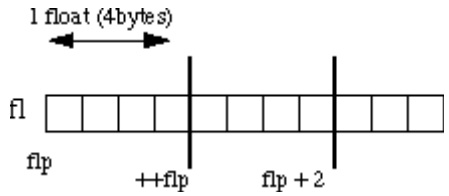


Fig. 9.2 Pointer Arithmetic Assume that flp points to fl then if we increment the pointer (++flp) it moves to the position shown 4 bytes on. If on the other hand we added 2 to the pointer then it moves 2 **float positions** *i.e* 8 bytes as shown in the Figure.

Pointer and Functions

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (*e.g.* `var` parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

The usual function *call*:

```
swap(a, b)
```
WON'T WORK.

Pointers provide the solution: *Pass the address of the variables to the functions and access address of function.*

Thus our function call in our program would look like this:

```
swap(&a, &b)
```

The Code to swap is fairly straightforward:

```
    void swap(int *px, int *py)

                   { int temp;

                               temp = *px;
                               /* contents of pointer */

                               *px = *py;
                               *py = temp;
                }
```

We can return pointer from functions. A common example is when passing back structures. *e.g.*:

```
typedef struct {float x,y,z;} COORD;

                main()

                 {  COORD p1, *coord_fn();
                                              /* declare fn to return ptr
of
                                               COORD type */

                                               ....
                                               p1 = *coord_fn(...);
                                  /* assign contents of address returned */
```

```
                                        ....
                                }

   COORD *coord_fn(...)

                {  COORD p;

                                        .....
                                        p = ....;
                                        /* assign structure values
*/

                                        return &p;
                                        /* return address of p */
                                }
```

Here we return a pointer whose contents are immediately *unwrapped* into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function ha squit though so this is perfectly safe.

Pointers and Arrays

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
   int a[10], x;
                int *pa;

                pa = &a[0];  /* pa pointer to address of a[0] */

                x = *pa;
                /* x = contents of pa (a[0] in this case) */
```

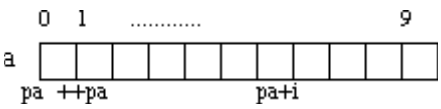


Fig. 9.3 Arrays and Pointers

To get somewhere in the array (Fig. 9.3) using a pointer we could do:

```
  pa + i ≡a[i]
```

WARNING: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its link between arrays and pointers.

For example we can just type

```
pa = a;
```

instead of

```
pa = &a[0]
```

and

a[i] can be written as *(a + i).
i.e. &a[i] ≡ a + i.

We also express pointer addressing like this:

pa[i] ≡ *(pa + i).

However pointers and arrays are different:

- A pointer is a variable. We can do pa = a and pa++.
- An Array is not a variable. a = pa and a++ ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this.

We can now understand how arrays are passed to functions.

When an array is passed to a function what is actually passed is its initial elements location in memory.

So:

strlen(s) ≡ strlen(&s[0])

This is why we declare the function:

```
int strlen(char s[]);
```

An equivalent declaration is : int strlen(char *s);
since char s[] ≡ char *s.

strlen() is a *standard library* function that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
                { char *p = s;
```

```
                                while (*p != `\0`);
                                  p++;
                                return p-s;
                    }
```

Now lets write a function to copy a string to another string. strcpy() is a standard library function that does this.

```
   void strcpy(char *s, char *t)

                    {  while ( (*s++ = *t++) != `\0`);}
```

This uses pointers and assignment by value.

Very Neat!!

NOTE: Uses of Null statements with while.

Arrays of Pointers

We can have arrays of pointers since pointers are variables.

Example use:

Sort lines of text of different length.

NOTE: Text can't be moved or compared in a single operation.

Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

How can we do this?:

- Store lines end-to-end in one big `char` array (Fig. 9.4). `\n` will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using `strcmp()` standard library function.
- If 2 lines are out of order -- swap pointer in pointer array (not text).

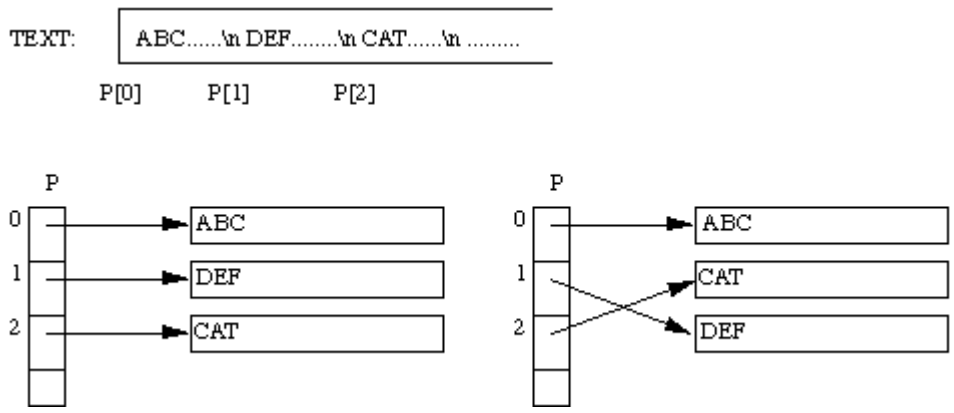


Fig. 9.4 Arrays of Pointers (String Sorting Example)

This eliminates:

- complicated storage management.
- high overheads of moving lines.

Multidimensional arrays and pointers

We should think of multidimensional arrays in a different way in C:

A 2D array is really a 1D array, each of whose elements is itself an array

Hence

`a[n][m]` notation.

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant.

The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

Consider`int a[5][35]` to be passed in a function:

We can do:

```
f(int a[][35]) {.....}
```

or even:

```
f(int (*a)[35]) {.....}
```

We need parenthesis (*a) since [] have a higher precedence than *

So:

`int (*a)[35];` declares a pointer to an array of 35 `int`s.

`int *a[35];` declares an array of 35 pointers to `int`s.

Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.

Consider:

```
char *name[10];

char Aname[10][20];
```

We can legally do `name[3][4]` and `Aname[3][4]` in C.

However

- `Aname` is a <u>true</u> 200 element 2D char array.
- access elements via
  *20*row + col + base_address*
  in memory.
- `name` has 10 pointer elements.

NOTE: If each pointer in `name` is set to point to a 20 element array then <u>and only then</u> will 200 chars be set aside (+ 10 elements).

The advantage of the latter is that each pointer can point to arrays be of different length.

Consider:

```
char *name[] = { ``no month'', ``jan'',
                          ``feb'', ... };
             char Aname[][15] = { ``no month'', ``jan'',
                          ``feb'', ... };
```

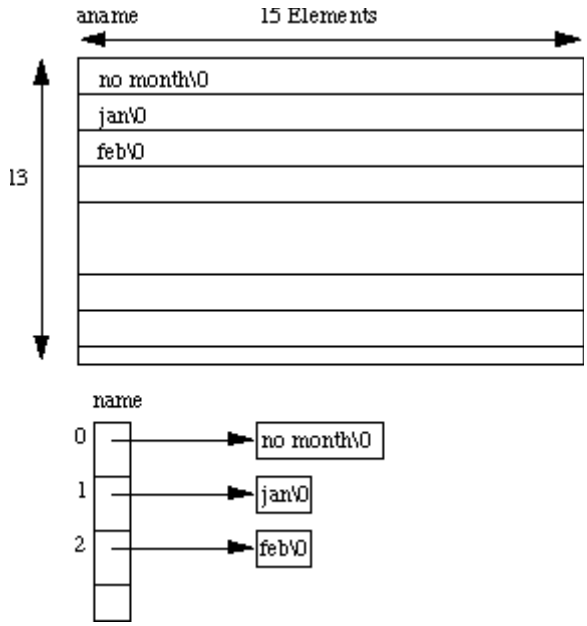


Fig. 2D Arrays and Arrays of Pointers

Static Initialisation of Pointer Arrays

Initialisation of arrays of pointers is an ideal application for an internal static array.

```
some_fn()
                { static char *months = { ``no month'',
                                          ``jan'', ``feb'',
                                          ...};

                                      }
```

`static reserves a private permanent bit of memory.`

Pointers and Structures

These are fairly straight forward and are easily defined. Consider the following:

```
   struct COORD {float x,y,z;} pt;
                struct COORD *pt_ptr;
```

```
pt_ptr = &pt; /* assigns pointer to pt */
```

the $->$ operator lets us access a member of the structure pointed to by a pointer.*i.e.*:

```
pt_ptr->x = 1.0;

pt_ptr->y = pt_ptr->y - 3.0;
```

Example: Linked Lists

```
typedef struct {  int value;
                                        ELEMENT *next;
                              } ELEMENT;



ELEMENT n1, n2;




n1.next = &n2;
```

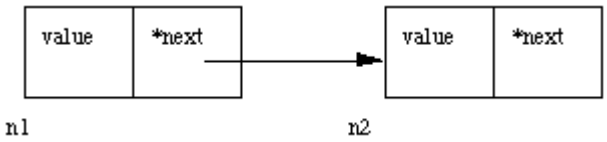


Fig. ❑ Linking Two Nodes NOTE: We can only declare `next as a pointer to ELEMENT. We cannot have a element of the variable type as this would set up a` *recursive* `definition which is` **NOT ALLOWED**`. We are allowed to set a pointer reference since 4 bytes are set aside for any pointer.`

`The above code links a node n1 to n2 (Fig. 9.6) we will look at this matter further in the next Chapter.`

Common Pointer Pitfalls

Here we will highlight two common mistakes made with pointers.

Not assigning a pointer to memory address before using it

```
int *x;

              *x = 100;
```

`we need a physical location say: int y;`

```
x = &y;
*x = 100;
```

This may be hard to spot. **NO COMPILER ERROR**. Also x could some random address at initialisation.

Illegal indirection

Suppose we have a function `malloc()` which tries to allocate memory dynamically (at run time) and returns a pointer to block of memory requested if successful or a `NULL` pointer otherwise.

`char *malloc()` -- a standard library function (see later).

Let us have a pointer: `char *p;`

Consider:

```
*p = (char *) malloc(100);   /* request 100 bytes of memory */
```

```
*p = `y';
```

There is mistake above. What is it?

No * in

```
*p = (char *) malloc(100);
```

Malloc returns a pointer. Also `p` does not point to any address.

The correct code should be:

```
p = (char *) malloc(100);
```

If code rectified one problem is if no memory is available and `p` is `NULL`. Therefore we can't do:

```
*p = `y';.
```

A good C program would check for this:

```
p = (char *) malloc(100);
              if ( p == NULL)
                                                  \
                { printf(``Error: Out of Memory   n'');
                                              exit(1);
```

```
                    }
        *p = `y';
```

Exercise

1. Write a C program to read through an array of any type using pointers. Write a C program to scan through this array to find a particular value.

2. Write a program to find the number of times that a given word(i.e. a short string) occurs in a sentence (i.e. a long string!).

Read data from standard input. The first line is a single word, which is followed by general text on the second line. Read both up to a newline character, and insert a terminating null before processing.

Typical output should be:

```
 The word is "the".
 The sentence is "the cat sat on the mat".
The word occurs 2 times.
```

3. Write a program that takes three variable (a, b, b) in as separate parameters and rotates the values stored so that value a goes to be, b, to c and c to a.

Files

Files are the most common form of a stream.

The first thing we must do is *open* a file. The function `fopen()` does this:

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

`fopen` returns a pointer to a FILE. The `name` string is the name of the file on disc that we wish to access. The `mode` string controls our type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned.

```
Modes include: ``r'' -- read,
               ``w'' -- write and
               ``a'' -- append.
```

To open a file we must have a stream (file pointer) that *points* to a `FILE` structure.

So to open a file, called *myfile.dat* for reading we would do:

```
  FILE *stream, *fopen();
                /* declare a stream and prototype fopen */

                stream = fopen(``myfile.dat'',``r'');
```

it is good practice to to check file is opened

correctly:

```
                if ( (stream = fopen( ``myfile.dat'',

                                        ``r'')) == NULL)

                                                \

                        {  printf(``Can't open %s n'',

                                        ``myfile.dat'');

                         exit(1);

                        }

                ......
```

Reading and writing files

The functions `fprintf` and `fscanf` a commonly used to access files.

```
int fprintf(FILE *stream, char *format, args..)
                int fscanf(FILE *stream, char *format, args..)
```

These are similar to `printf and scanf except that data is read from the` *`stream`* `that must have been opened with fopen().`

`The stream pointer is automatically incremented with ALL file read/write functions. We` **`do not`** `have to worry about doing this.`

```
  char *string[80]
                FILE *stream, *fopen();
```

```
                if ( (stream = fopen(...)) != NULL)
                                fscanf(stream,``%s'', string);
```

Other functions for files:

int getc(FILE *stream), int fgetc(FILE *stream)

int putc(char ch, FILE *s), int fputc(char ch, FILE *s)

These are like getchar, putchar.

getc is defined as preprocessor MACRO in stdio.h. fgetc is a C library function. Both achieve the same result!!

   fflush(FILE *stream) -- flushes a stream.

   fclose(FILE *stream) -- closes a stream.

We can access predefined streams with fprintf *etc.*

```
   fprintf(stderr,``Cannot Compute!! \n'');

    fscanf(stdin,``%s'',string);
```

sprintf and sscanf

These are like `fprintf` and `fscanf` except they read/write to a string.

int sprintf(char *string, char *format, args..)

int sscanf(char *string, char *format, args..)

For Example:

```
  float full_tank = 47.0; /* litres */
                float miles = 300;
                char miles_per_litre[80];

                sprintf( miles_per_litre,``Miles per litre
                                = %2.3f'', miles/full_tank);
```

Stream Status Enquiries

There are a few useful stream enquiry functions, prototyped as follows:

```
 int feof(FILE *stream);
 int ferror(FILE *stream);
```

```
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

Their use is relatively simple:

`feof()`
: -- returns true if the stream is currently at the end of the file. So to read a stream,`fp`, line by line you could do:
```
while ( !feof(fp) )
  fscanf(fp,"%s",line);
```

`ferror()`
: -- reports on the error state of the stream and returns true if an error has occurred.

`clearerr()`
: -- resets the error indication for a given stream.

`fileno()`
: -- returns the integer file descriptor associated with the named stream.

Low Level I/O

This form of I/O is UNBUFFERED -- each read/write request results in accessing disk (or device) directly to fetch/put a specific number of **bytes**.

There are no formatting facilities -- we are dealing with bytes of information.

This means we are now using binary (and not text) files.

Instead of file pointers we use *low level* `file handle or file descriptors` which give a unique integer number to identify each file.

To Open a file use:

`int open(char *filename, int flag, int perms)` -- this returns a file descriptor or -1 for a **fail**.

The `flag` controls file access and has the following predefined in `fcntl.h`:

`O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY` + others see online `man` pages or reference manuals.

`perms` -- best set to 0 for most of our applications.

The function:

```
creat(char *filename, int perms)
```

can also be used to create a file.

`int close(int handle)` -- close a file

```
int read(int handle, char *buffer,
unsigned length)

int write(int handle, char *buffer, unsigned length)
```

are used to read/write a specific number of bytes from/to a file (handle) stored or to be put in the memory location specified by `buffer`.

The `sizeof()` function is commonly used to specify the `length`.

read and write return the number of bytes read/written or -1 if they fail.

```
/* program to  read a list of floats from a binary file */
/* first byte of file is an integer saying how many  */
/* floats in file. Floats follow after it, File name got from */
/* command line */

#include<stdio.h>
#include<fcntl.h>

float bigbuff[1000];

main(int argc, char **argv)

{  int fd;
                int bytes_read;
                int file_length;

                if ( (fd = open(argv[1],O_RDONLY)) = -1)
                                { /* error file not open */....
                                                perror("Datafile");
                                                exit(1);
                                }
                if ( (bytes_read = read(fd,&file_length,
                                                sizeof(int))) == -1)
                                { /* error reading file */...
                                                exit(1);
                                }
                if ( file_length > 999 )  {/* file too big */ ....}
                if ( (bytes_read = read(fd,bigbuff,
                                                file_length*sizeof(float)))
== -1)
                                { /* error reading open */...
                                                exit(1);
                                }
}
```

Exercises

1. Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.

Copy the file a block (512 bytes) at a time.

```
Check:  that the program has two arguments
                 or print "Program need two arguments"
               that the first name file is readable
                 or print "Cannot open file .... for reading"
                that the second file is writable
                  or print "Cannot open file .... for writing"
```

2. Write a program last that prints the last *n* lines of a text file, by *n* and the file name should be specified form command line input. By default *n* should be 5, but your program should allow an optional argument so that

```
last -n file.txt
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

3. Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.