Roinn na Matamaitice Feidhmí

Coláiste na hOllscoile Corcaigh

A COURSE IN C PROGRAMMING

Diarmuid O' Ríordáin, BE, MEngSc, MIEI

Department of Applied Mathematics

University College Cork

6th Revision - 12/02/2016

Chapter 1

Introduction

1.1 Origin of C

The C Programming Language was initially developed by Denis Ritchie using a Unix system in 1972. This was varied and modified until a standard was defined by Brian Kernighan and Dennis Ritchie in 1978 in "The C Programming Language".
By the early 80's many versions of C were available which were inconsistent with each other in many aspects. This led to a standard being defined by ANSI in 1983. It is this standard this set of notes primarily addresses.

Why use C ?

Industry Presence : Over the last decade C has become one of the most widely used development languages in the software industry. Its importance is not entirely derived from its use as a primary development language but also because of its use as an interface language to some of the newer “visual” languages and of course because of its relationship with C++.

Middle Level : Being a Middle level language it combines elements of high level languages with the functionality of assembly language. C supports data types and operations on data types in much the same way as higher level languages as well as allowing direct manipulation of bits, bytes, words and addresses as is possible with low level languages.

Portability : With the availability of compilers for almost all operating systems and hardware platforms it is easy to write code on one system which can be easily ported to another as long as a few simple guidelines are followed.

Flexibility : Supporting its position as the mainstream development language C can be interfaced readily to other programming languages.

Malleable : C, unlike some other languages, offers little restriction to the programmer with regard to data types -- one type may be coerced to another type as the situation dictates. However this *feature* can lead to sloppy coding unless the programmer is fully aware of what rules are being bent and why.

Speed : The availability of various optimising compilers allow extremely efficient code to be generated automatically.

1.2 The “Hello World” Program

A C program consists of one or more functions or code modules. These are essentially groups of instructions that are to be executed as a unit in a given order and that can be referenced by a unique name. Each C program must contain a ***main()*** function. This is the first function called when the program starts to run. Note that while "main" is not a C keyword and hence not reserved it should be used only in this context.

A C program is traditionally arranged in the following order but not strictly as a rule.

Function prototypes and global data declarations
The main() function
Function definitions

Consider first a simple C program which simply prints a line of text to the computer screen. This is traditionally the first C program you will see and is commonly called the “Hello World” program for obvious reasons.

```
#include <stdio.h>

void main()
{
/* This is how comments are implemented in C
           to comment out a block of text */
// or like this for a single line comment

printf( "Hello World\n" ) ;

}
```

As you can see this program consists of just one function the mandatory *main* function. The parentheses, ( ), after the word main indicate a function while the curly braces, { }, are used to denote a block of code -- in this case the sequence of instructions that make up the function.

Comments are contained within a /* ... */ pair in the case of a block comment or a double forward slash, //, may be used to comment out the remains of a single line of test.

The line

```
printf("Hello World\n " ) ;
```

is the only C statement in the program and <u>must</u> be terminated by a semi-colon.

The statement calls a function called `printf` which causes its argument, the string of text within the quotation marks, to be printed to the screen. The characters \n are not printed as these characters are interpreted as special characters by the printf function in this case printing out a newline on the screen. These characters are called *escape sequences* in C and cause special actions to occur and are preceded always by the backslash character, \ .

All C compiler include a library of standard C functions such as printf which allow the programmer to carry out routine tasks such as I/O, maths operations, etc. but which are not part of the C language, the compiled C code merely being provided with the compiler in a standard form.

Header files must be included which contain *prototypes* for the standard library functions and declarations for the various variables or constants needed. These are normally denoted by a .h extension and are processed automatically by a program called the *Preprocessor* prior to the actual compilation of the C program.

The line

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

instructs the preprocessor to include the file stdio.h into the program before compilation so that the definitions for the standard input/output functions including printf will be present for the compiler. The angle braces denote that the compiler should look in the default "INCLUDE" directory for this file. A pair of double quotes indicate that the compiler should search in the specified path e.g.

```
    #include "d:\myfile.h"
```

NB : C is case sensitive i.e. `printf()` and `Printf()` would be regarded as two different functions.

1.3 The C Programming Environment

Program development is nowadays carried out in specifically designed software systems or workbenches with editing, compilation, linking, debugging and execution facilities built in. In this course we will be making use of a Microsoft system but the features found in this are to be found in one form or another in almost all modern systems.

The first phase of development involves the creation and editing of a file containing the appropriate C instructions which will be stored using a file extension of **.c** normally to invoke the C compiler, e.g. fname.c.

The next step is to take the C program and to compile it into object code or machine language code. The C compiler includes the aforementioned preprocessor which is called automatically before the code translation takes place. This preprocessor acts on special commands or directives from the programmer to manipulate the text of the C code before compilation commences. These directives might involve including other source files in the file to be compiled, replacing special symbols with specific replacement text, etc. Once this is done the C code is translated into object code and stored in a file with the extension **`.obj`**, e.g. `fname.obj`.

The final phase in building the executable program is called linking. After the compilation stage the C code has been translated into machine recognisable code but is in a somewhat unconnected state. The program invariably contains references to standard library functions or functions contained in other libraries or modules which must be connected to the C program at link time. This simply involves linking the machine code for these functions with the program's object code to complete the build process and produce an executable file with an extension **`.exe`** e.g. `fname.exe.`

The executable program can be loaded and run from within the programming environment itself or may be run from the host environment directly. If it executes as expected that is the end of the task. However if this does not happen it may require the use of the debugger to isolate any logical problems. The debugger allows us to step through the code instruction by instruction or up to predefined break-points and to look at the values of variables in the code in order to establish where errors are introduced.

Chapter 2

Variables, Data Types, I/O and Operators

In order to be useful a program must be able to represent real life quantities or data e.g. a person's name, age, height, bank balance, etc. This data will be stored in memory locations called variables that we will name ourselves. However so that the data may be represented as aptly as possible the variables will have to be of different types to suit their data. For example while an integer can represent the age of a person reasonably well it won't be able to represent the pounds and pence in a bank balance or the name of an individual quite so well.

2.1 Basic Data Types

There are five basic data types char, int, float, double, and void. All other data types in C are based on these. Note that the size of an int depends on the standard size of an integer on a particular operating system.

char	1 byte (8 bits) with range -128 to 127
int	16-bit OS : 2 bytes with range -32768 to 32767 32-bit OS : 4 bytes with range -2,147,483,648 to 2,147,483,647
float	4 bytes with range 10^{-38} to 10^{38} with 7 digits of precision
double	8 bytes with range 10^{-308} to 10^{308} with 15 digits of precision
void	generic pointer, used to indicate no function parameters etc.

Modifying Basic Types

Except for type ***`void`*** the meaning of the above basic types may be altered when combined with the following keywords.

```
signed
unsigned
long
short
```

The *`signed`* and *`unsigned`* modifiers may be applied to types char and int and will simply change the range of possible values. For example an *`unsigned char`* has a range of 0 to 255, all positive, as opposed to a signed char which has a range of -128 to 127. An *unsigned integer* on a 16-bit system has a range of 0 to 65535 as opposed to a *signed int* which has a range of -32768 to 32767.

Note however that the default for type int or char is signed so that the type *signed char* is always equivalent to type *char* and the type *signed int* is always equivalent to *int*.

The *`long`* modifier may be applied to type int and double only. A *`long int`* will require 4 bytes of storage no matter what operating system is in use and has a range of -2,147,483,648 to 2,147,483,647. A *`long double`* will require 10 bytes of storage and will be able to maintain up to 19 digits of precision.

The `short` modifier may be applied only to type *int* and will give a 2 byte integer independent of the operating system in use.

NB : Note that the keyword *int* may be omitted without error so that the type *unsigned* is the same as type `unsigned int`, the type `long` is equivalent to the type `long int`, and the type `short` is equivalent to the type `short int`.

2.2 Variables

A variable is a named piece of memory which is used to hold a value which may be modified by the program. A variable thus has three attributes that are of interest to us : its **type,** its **value** and its **address.**

The variable's type informs us what type and range of values it can represent and how much memory is used to store that value. The variable's address informs us where in memory the variable is located (which will become increasingly important when we discuss pointers later on).

All C variables must be declared as follows :-

```
type variable-list ;
```

For Example :-

```
int i ;
char a, b, ch ;
```

Variables are declared in three general areas in a C program.

When declared inside functions as follows they are termed **local** variables and are visible (or accessible) within the function (or code block) only.

```
void main()
{
int i, j ;
...
}
```

A local variable is created i.e. allocated memory for storage upon entry into the code block in which it is declared and is destroyed i.e. its memory is released on exit. This means that values cannot be stored in these variables for use in any subsequent calls to the function .

When declared outside functions they are termed **global** variables and are visible throughout the file or have file scope. These variables are created at program start-up and can be used for the lifetime of the program.

```
int i ;
void main()
{
...
}
```

When declared within the braces of a function they are termed the formal parameters of the function as we will see later on.

```
int func1( int a, char b ) ;
```

Variable Names

Names of variables and functions in C are called identifiers and are case sensitive. The first character of an identifier must be either a letter or an underscore while the remaining characters may be letters, numbers, or underscores. Identifiers in C can be up to 31 characters in length.

Initialising Variables

When variables are declared in a program it just means that an appropriate amount of memory is allocated to them for their exclusive use. This memory however is **not initialised** to zero or to any other value automatically and so will contain random values unless specifically initialised before use.

***Syntax* :- `type var-name = constant ;`**

For Example :-

```
      char ch = 'a' ;
      double d = 12.2323 ;
      int i, j = 20 ; /* note in this case  i is not initialised
*/
```

Storage Classes

There are four storage class modifiers used in C which determine an identifier's storage duration and scope.

```
                  auto
                  static
                  register
                  extern
```

An identifier's storage duration is the period during which that identifier exists in memory. Some identifiers exist for a short time only, some are repeatedly created and destroyed and some exist for the entire duration of the program. An identifier's scope specifies what sections of code it is accessible from.

The auto storage class is implicitly the default storage class used and simply specifies a normal local variable which is visible within its own code block only and which is created and destroyed automatically upon entry and exit respectively from the code block.

The register storage class also specifies a normal local variable but it also requests that the compiler store a variable so that it may be accessed as quickly as possible, possibly from a CPU register.

The static storage class causes a local variable to become permanent within its own code block i.e. it retains its memory space and hence its value between function calls.

When applied to global variables the static modifier causes them to be visible only within the physical source file that contains them i.e. to have file scope. Whereas the extern modifier which is the implicit default for global variables enables them to be accessed in more than one source file.

For example in the case where there are two C source code files to be compiled together to give one executable and where one specific global variable needs to be used by both the extern class allows the programmer to inform the compiler of the existence of this global variable in both files.

Constants

Constants are fixed values that cannot be altered by the program and can be numbers, characters or strings.

Some Examples :-

```
char :  'a', '$', '7'
int :  10, 100, -100
unsigned :  0, 255
float :  12.23456, -1.573765e10, 1.347654E-13
double :  1433.34534545454, 1.35456456456456E-200
long :  65536, 2222222
string : “Hello World\n”
```

NB : Floating point constants default to type double. For example the following code segment will cause the compiler to issue a warning pertaining to floating point conversion in the case of `f_val` but not in the case of `d_val`..

```
float f_val ;
double d_val ;
f_val = 123.345 ;
d_val = 123.345 ;
```

However the value may be coerced to type float by the use of a modifier as follows :-

```
f = 123.345F ;
```

Integer constants may also be forced to be a certain type as follows :-

```
100U --- unsigned
100L --- long
```

Integer constants may be represented as either decimal which is the default, as hexadecimal when preceded by "0x", e.g. 0x2A, or as octal when preceded by "O", e.g. O27.

Character constants are normally represented between single quotes, e.g. 'a', 'b', etc. However they may also be represented using their ASCII (or decimal) values e.g. 97 is the ASCII value for the letter 'a', and so the following two statements are equivalent. (See Appendix A for a listing of the first 128 ASCII codes.)

```
char ch = 97 ;
char ch = 'a' ;
```

There are also a number of special character constants sometimes called *Escape Sequences*, which are preceded by the backslash character '\', and have special meanings in C.

\n	newline
\t	tab
\b	backspace
\'	single quote
\"	double quote
\0	null character
\xdd	represent as hexadecimal constant

2.3 Console Input / Output

This section introduces some of the more common input and output functions provided in the C standard library.

printf()

The printf() function is used for formatted output and uses a control string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required. The more common format specifiers are given below

%c	character	%f	floating point
%d	signed integer	%lf	double floating point
%i	signed integer	%e	exponential notation
%u	unsigned integer	%s	string
%ld	signed long	%x	unsigned hexadecimal
%lu	unsigned long	%o	unsigned octal
		%%	prints a % sign

For Example :-
```
            int i ;

            printf(  "%d", i ) ;
```

The `printf()` function takes a variable number of arguments. In the above example two arguments are required, the format string and the variable *i*. The value of *i* is substituted for the format specifier %d which simply specifies how the value is to be displayed, in this case as a signed integer.

Some further examples :-

```
int i = 10, j = 20 ;
char ch = 'a' ;
double f = 23421.2345 ;

printf( "%d + %d", i, j ) ;  /* values are substituted from
                              the variable list  in order as required  */
printf( "%c", ch ) ;

printf( "%s", "Hello World\n" ) ;

printf( "The value of f is : %lf", f ) ;/*Output as : 23421.2345
*/
printf( "f in exponential form : %e", f ) ; /*   Output   as   :
2.34212345e+4
```

Field Width Specifiers

Field width specifiers are used in the control string to format the numbers or characters output appropriately .

Syntax :- **%[total width printed][.decimal places printed]format specifier**

where square braces indicate optional arguments.

For Example :-
```
int i = 15 ;
float f = 13.3576 ;
```

```
printf( "%3d", i ) ;   /* prints "_15 " where _ indicates a space
                          character */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width
                          of 6 and displays 2 decimal places  */
printf( "%*.*f", 6,2,f ) ;   /* prints  "_13.36" as above. Here *
is used as replacement character for field widths     */
```

There are also a number of flags that can be used in conjunction with field width specifiers to modify the output format. These are placed directly after the % sign. A - (minus sign) causes the output to be left-justified within the specified field, a + (plus sign) displays a plus sign preceding positive values and a minus preceding negative values, and a 0 (zero) causes a field to be padded using zeros rather than space characters.

scanf()

This function is similar to the printf function except that it is used for formatted input. The format specifiers have the same meaning as for `printf()` and the space character or the newline character are normally used as delimiters between different inputs.

For Example :-

```
int i, d ;
char c ;
float f ;

scanf( "%d", &i ) ;

scanf( "%d %c %f", &d, &c, &f ) ; /* e.g. type "10_x_1.234RET" */

scanf( "%d:%c", &i, &c ) ;         /* e.g.  type "10:xRET"  */
```

The & character is the *address of* operator in C, it returns the address in memory of the variable it acts on. (Aside : This is because C functions are nominally call--by--value. Thus in order to change the value of a calling parameter we must tell the function exactly where the variable resides in memory and so allow the function to alter it directly rather than to uselessly alter a copy of it.)

Note that while the space and newline characters are normally used as delimiters between input fields the actual delimiters specified in the format string of the scanf statement must be reproduced at the keyboard faithfully as in the case of the last sample call. If this is not done the program can produce somewhat erratic results!

The scanf function has a return value which represents the number of fields it was able to convert successfully.

For Example :-

```
          num = scanf( "%c %d", &ch, &i );
```

This scanf call requires two fields, a character and an integer, to be read in so the value placed in *num* after the call should be 2 if this was successful. However if the input was “a bc” then the first character field will be read correctly as ‘a’ but the integer field will not be converted correctly as the function cannot reconcile “bc” as an integer. Thus the function will return 1 indicating that one field was successfully converted. Thus to be safe the return value of the scanf function should be checked always and some appropriate action taken if the value is incorrect.

getchar() and putchar()

These functions are used to input and output single characters. The *getchar()* function reads the ASCII value of a character input at the keyboard and displays the character while *putchar()* displays a character on the standard output device i.e. the screen.

For Example :-
```
            char ch1, ch2 ;

            ch1 = getchar() ;

            ch2 = 'a' ;
            putchar( ch2 ) ;
```

NB : The input functions described above, `scanf()` and `getchar()` are termed buffered input functions. This means that whatever the user types at the keyboard is first stored in a data buffer and is not actually read into the program until either the buffer fills up and has to be flushed or until the user flushes the buffer by hitting **RET** whereupon the required data is read into the program. The important thing to remember with buffered input is that no matter how much data is taken into the buffer when it is flushed the program just reads as much data as it needs from the start of the buffer allowing whatever else that may be in the buffer to be discarded.

For Example :-
```
      char ch1, ch2;

      printf( "Enter two characters : " ) ;
      ch1 = getchar() ;
      ch2 = getchar() ;
      printf( "\n The characters are %c and %c\n", ch1, ch2 ) ;
```

In the above code segment if the input is "abcdef**RET**" the first two characters are read into the variables all the others being discarded, but control does not return to the program until the **RET** is hit and the buffer flushed. If the input was "a**RET**" then a would be placed in ch1 and **RET** in ch2.

_flushall()

The `_flushall` function writes the contents of all output buffers to the screen and clears the contents of all input buffers. The next input operation (if there is one) then reads new data from the input device into the buffers.

This function should be used always in conjunction with the buffered input functions to clear out unwanted characters from the buffer <u>**after each**</u> input call.

getch() and getche()

These functions perform the same operation as getchar() except that they are unbuffered input functions i.e. it is not necessary to type **RET** to cause the values to be read into the program they are read in immediately the key is pressed. getche() echoes the character hit to the screen while getch() does not.

For example :-
```
            char ch ;
            ch = getch() ;
```

2.4 Operators

One of the most important features of C is that it has a very rich set of built in operators including arithmetic, relational, logical, and bitwise operators.

Assignment Operator

```
            int x ;
            x = 20 ;
```

Some common notation :- lvalue -- left hand side of an assignment operation
rvalue -- right hand side of an assignment operation

Type Conversions :- the value of the right hand side of an assignment is converted to the type of the lvalue. This may sometimes yield compiler warnings if information is lost in the conversion.

For Example :-
```
int x ;
char ch ;
float f ;

ch = x ;         /*  ch  is  assigned  lower  8  bits  of  x,  the
remaining bits are discarded so we have a possible information
loss  */
x = f ;          /* x is assigned non fractional part of f only
within int range, information loss possible  */
f = x ;          /* value of x is converted to floating point */
```

Multiple assignments are possible to any degree in C, the assignment operator has right to left associativity which means that the rightmost expression is evaluated first.

For Example :-
```
            x = y = z = 100 ;
```

In this case the expression `z = 100` is carried out first. This causes the value 100 to be placed in z with the value of the whole expression being 100 also. This expression value is then taken and assigned by the next assignment operator on the left i.e. `x = y = ( z = 100 ) ;`

Arithmetic Operators

`+ - * /` --- same rules as mathematics with * and / being evaluated before + and -.
% -- modulus / remainder operator

For Example :-
```
int a = 5, b = 2, x ;
float c = 5.0, d = 2.0, f ;

x = a / b ;      //  integer division, x = 2.
f = c / d  ;     //  floating point division, f = 2.5.
x = 5 % 2 ;      //  remainder operator, x = 1.

x = 7 + 3 * 6 / 2 - 1 ;// x=15,* and / evaluated ahead of + and -
.
```

Note that parentheses may be used to clarify or modify the evaluation of expressions of any type in C in the same way as in normal arithmetic.

```
x = 7 + ( 3 * 6 / 2 ) - 1 ;//  clarifies  order  of  evaluation
without penalty
x = ( 7 + 3 ) * 6 / ( 2 - 1 ) ;  // changes order of evaluation,
x = 60 now.
```

Increment and Decrement Operators

There are two special unary operators in C, Increment ++, and Decrement -- , which cause the variable they act on to be incremented or decremented by 1 respectively.

For Example :-

```
           x++ ;      /* equivalent to      x = x + 1 ;    */
```

++ and -- can be used in prefix or postfix notation. In prefix notation the value of the variable is either incremented or decremented and is then read while in postfix notation the value of the variable is read first and is then incremented or decremented.

For Example :-

```
int i,  j = 2 ;

i = ++ j  ;      /* prefix  :-   i has value 3, j has value 3  */
i = j++ ;  /* postfix  :-  i  has value 3, j has value 4   */
```

Special Assignment Operators

Many C operators can be combined with the assignment operator as shorthand notation

For Example :-

```
           x = x + 10 ;
```

can be replaced by

```
           x += 10 ;
```

Similarly for `-=,  *=,   /=,  %=`, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

Relational Operators

The full set of relational operators are provided in shorthand notation

```
     >     >=    <     <=    ==    !=
```

For Example :-

```
           if ( x == 2 )
                printf( "x is equal to 2\n" ) ;
```

Logical Operators

&&	--	Logical AND
||	--	Logical OR
!	--	Logical NOT

For Example :-
```
if (  x >= 0 && x < 10  )
  printf( " x is greater than or equal to zero and less than ten.\n" ) ;
```

NB : There is no Boolean type in C so TRUE and FALSE are deemed to have the following meanings.

FALSE	--	value zero
TRUE	--	any non-zero value but 1 in the case of in-built relational operations

For Example :-

2 > 1	-- TRUE so expression has value 1
2 > 3	-- FALSE so expression has value 0
i = 2 > 1 ;	-- relation is TRUE -- has value 1, i is assigned value 1

NB : Every C expression has a value. Typically we regard expressions like 2 + 3 as the only expressions with actual numeric values. However the relation 2 > 1 is an expression which evaluates to TRUE so it has a value 1 in C. Likewise if we have an expression x = 10 this has a value which in this case is 10 the value actually assigned.

NB : Beware of the following common source of error. If we want to test if a variable has a particular value we would write for example

```
if ( x == 10 )  …
```

But if this is inadvertently written as

```
if ( x = 10 ) …
```

this will give no compilation error to warn us but will compile and assign a value 10 to x when the condition is tested. As this value is non-zero the if condition is deemed true no matter what value x had originally. Obviously this is possibly a serious logical flaw in a program.

Bitwise Operators

These are special operators that act on **char or int arguments only**. They allow the programmer to get closer to the machine level by operating at bit-level in their arguments.

&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR	~	Ones Complement
>>	Shift Right	<<	Shift left

Recall that type char is one byte in size. This means it is made up of 8 distinct bits or binary digits normally designated as illustrated below with Bit 0 being the Least Significant Bit (LSB) and Bit 7 being the Most Significant Bit (MSB). The value represented below is 13 in decimal.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	0	1

An integer on a 16 bit OS is two bytes in size and so Bit 15 will be the MSB while on a 32 bit system the integer is four bytes in size with Bit 31 as the MSB.

Bitwise AND, &

RULE : If any two bits in the same bit position are set then the resultant bit in that position is set otherwise it is zero.

For Example :-

```
          1011 0010      (178)
&         0011 1111      (63)
=         0011 0010      (50)
```

Bitwise OR, |

RULE : If either bit in corresponding positions are set the resultant bit in that position is set.

For Example :-

```
          1011 0010             (178)
 |        0000 1000             (63)
=         1011 1010             (186)
```

Bitwise XOR, ^

RULE : If the bits in corresponding positions are different then the resultant bit is set.

For Example :-

```
          1011 0010             (178)
^             0011 1100          (63)
=             1000 1110         (142)
```

Shift Operators, << and >>

RULE : These move all bits in the operand left or right by a specified number of places.

Syntax :

```
variable << number of places
variable >> number of places
```

For Example :-

```
2 << 2 = 8
```

i.e.

0000 0010 becomes 0000 1000

NB : shift left by one place multiplies by 2
shift right by one place divides by 2

Ones Complement

RULE : Reverses the state of each bit.

For Example :-

1101 0011 becomes 0010 1100

NB : With all of the above bitwise operators we must work with decimal, octal, or hexadecimal values as binary is not supported directly in C.

The bitwise operators are most commonly used in system level programming where individual bits of an integer will represent certain real life entities which are either on or off, one or zero. The programmer will need to be able to manipulate individual bits directly in these situations.

A *mask* variable which allows us to ignore certain bit positions and concentrate the operation only on those of specific interest to us is almost always used in these situations. The value given to the mask variable depends on the operator being used and the result required.

For Example :- To clear bit 7 of a char variable.

```
char ch = 89 ;          // any value

char mask = 127 ;       // 0111 1111

ch = ch & mask ; // or  ch &= mask ;
```

For Example :- To set bit 1 of an integer variable.

```
int i = 234 ;           // any value

int mask = 2 ;          // a 1 in bit position 2

i |= mask ;
```

Implicit & Explicit Type Conversions

Normally in mixed type expressions all operands are converted **temporarily** up to the type of the largest operand in the expression.

Normally this automatic or implicit casting of operands follows the following guidelines in ascending order.

long double
double
float
unsigned long
long
unsigned int
signed int

For Example :-

```
int i ;
float f1, f2 ;

f1 = f2 + i ;
```

Since f2 is a floating point variable the value contained in the integer variable is temporarily converted or ***cast to*** a floating point variable also to standardise the addition operation in this case. However it is important to realise that no permanent modification is made to the integer variable.

Explicit casting coerces the expression to be of specific type and is carried out by means of the **cast operator** which has the following syntax.

Syntax : **`( type ) expression`**

For Example if we have an integer x, and we wish to use floating point division in the expression x/2 we might do the following

```
( float ) x  /  2
```

which causes x to be temporarily cast to a floating point value and then implicit casting causes the whole operation to be floating point division.

The same results could be achieved by stating the operation as

```
x  /  2.0
```

which essentially does the same thing but the former is more obvious and descriptive of what is happening.

NB : It should be noted that all of these casting operations, both implicit and explicit, require processor time. Therefore for optimum efficiency the number of conversions should be kept to a minimum.

Sizeof Operator

The sizeof operator gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types as we will see later on).

The expression is either an identifier or a type-cast expression (a type specifier enclosed in parentheses).

Syntax : ***`sizeof ( expression )`***

For Example :-

```
int x , size ;

size = sizeof ( x ) ;
printf("The integer x requires %d bytes on this machine", size);

printf( "Doubles take up %d bytes on this machine", sizeof (
double ) ) ;
```

Precedence of Operators

When several operations are combined into one C expression the compiler has to rely on a strict set of precedence rules to decide which operation will take preference. The precedence of C operators is given below.

Precedence	**Operator**	**Associativity**
Highest	( ) [ ] -> .	left to right
	! ~ ++ -- +(unary) -(unary) (type) * & sizeof	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= &= ^= |= <<= >>=	right to left
Lowest	,	left to right

Operators at the top of the table have highest precedence and when combined with other operators at the same expression level will be evaluated first.

For example take the expression

```
2 + 10 * 5 ;
```

Here * and + are being applied at the same level in the expression but which comes first ? The answer lies in the precedence table where the * is at a higher level than the + and so will be applied first.

When two operators with the same precedence level are applied at the same expression level the associativity of the operators comes into play.

For example in the expression

```
2 + 3 - 4 ;
```

the + and - operators are at the same precedence level but associate from left to right and so the addition will be performed first. However in the expression

```
x = y = 2 ;
```

as we have noted already the assignment operator associates from right to left and so the rightmost assignment is first performed.

NB : As we have seen already parentheses can be used to supersede the precedence rules and force evaluation along the lines we require. For example to force the addition in **2 + 10 * 5 ;** to be carried out first we would write it as **(2 + 10) * 5;**

2.5 Type Overflow & Underflow

When the value to be stored in a variable of a particular type is larger than the range of values that type can hold we have what is termed type overflow. Likewise when the value is smaller than the range of values the type can hold we have type underflow.

Overflow and underflow are only a problem when dealing with integer arithmetic. This is because C simply ignores the situation and continues on as if nothing had happened. With signed integer arithmetic adding two large positive numbers, the result of which will be larger than the largest positive signed int, will lead to a negative value being returned as the sign bit will be overwritten with data.

The situation is not quite so bad with unsigned integer arithmetic. Here all values are forced to be within range which will of course cause problems if you don’t expect overflow to occur. Adding 1 to the largest unsigned integer will give 0.

The unfortunate aspect of the matter however is that we cannot check for overflow until it has occurred. There are a number of ways to do this. For example when performing integer addition you might check the result by subtracting one of the operands from the result to see if you get the other. On the other hand you might subtract one operand from the largest integer to see if the result is greater than the second operand. If it is you know your operation will succeed. However the major flaw with these methods is that we are reducing the overall efficiency of the program with extra operations.

In general the optimum method for dealing with situations where overflow or underflow is possible is to use type long over the other integer types and inspect the results. Operations using long operands are in general slower than those using int operands but if overflow is a problem it is still a better solution than those mentioned above.

Floating point overflow is not a problem as the system itself is informed when it occurs which causes your program to terminate with a run-time error. If this happens you need to promote the variables involved in the offending operation to the largest possible and try again.

NB : The C standard library includes a number of exception handling functions to allow you to intercept these situations in your program.

2.6 Exercises

1. Write a program to check the sizes of the main C data types on your machine.

2. Write a program to illustrate whether the printf() standard library function truncates or rounds when printing out a floating point number.

3. Write a program to check what the following code segment outputs and explain the results.

```
            char c ;
            printf("sizeof( c ) = %d\n", sizeof( c ) ) ;
            printf("sizeof( 'a' ) = %d\n", sizeof( 'a' ) ) ;
            printf("sizeof( c = 'a' ) = %d\n", sizeof( c='a' ) )
;
```

4. Write a program which reads a character from the keyboard and writes out its ASCII representation.

Now write a program which reads in an integer from the keyboard and print out its character representation. Make certain you carry out appropriate bounds / error checking.

5. Use the getchar() function to read in a single character and output it to the screen e.g.

```
puts("Enter a character");
c = getchar() ;
printf("The character was %c\n",c);
```

Add another `c = getchar()` statement immediately after the existing one and explain what happens.

Repeat the above using the `getch()` function in place of `getchar()`.

6. Describe the output from each of the following statements.

```
i.  printf( "%-10d\n", 10000 ) ;
ii. printf( "%8.3f\n", 23.234 ) ;
iii.printf( "%+*.*lf\n", 10, 3, 1234.234 ) ;
iv. printf( "%x\n", 16 ) ;
v.  printf( "%10.3E", 234.65343 ) ;
```

7. Write down appropriate C statements to do the following.

i. Print a long int, 400000L, left justified in a 10 digit field padding it out with zeros if possible.
ii. Read a time of the form **hh:mm:ss** storing the parts of the time in integer variables *hour*, *minute*, and *second*. Skip the colons in the input field.
iii. Print out the following sequence of characters "**%, \ and " require special treatment**" .
iv. Read the value 123456789.012345456789e+5 into a floating point, a double and a long double variable and print all three out again with the maximum precision possible.

8. What value does x contain after each of the following where x is of type float.

```
i.  x = 7 + 3 * 6 / 2 - 1 ;
ii. x = 2 % 2 + 2 * 2 - 2 / 2 ;
iii.x = ( 3  * 9 * ( 3 + ( 4 * 5 / 3 ) ) ) ;
iv. x = 12.0 + 2 / 5 * 10.0 ;
v.  x = 2 / 5 + 10.0 * 3 - 2.5 ;
vi. x = 15 > 10 && 5 < 2 ;
```

9. Write a program to read Fahrenheit temperatures and print them in Celsius. The formula is °C = (5/9)(°F - 32). Use variables of type double in your program.

10. Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the value 3.14159 for "pi".

Chapter 3

Statements

3.1 Expressions and Statements

As mentioned at the outset every C program consists of one or more functions which are just groups of instructions that are to be executed in a given order. These individual instructions are termed ***statements*** in C.

We have already seen some simple examples of C statements when introducing the set of C operators. For example

```
x = 0 ;
```

is a simple statement that initialises a variable x to zero using the assignment operator. The statement is made up of two parts : the assignment operation and the terminating semi-colon. The assignment operation here is termed an ***expression*** in C. In general an expression consists of one of C's operators acting on one or more operands. To convert an expression into a C statement requires the addition of a terminating semi-colon.

A function call is also termed an expression. For example in the *hello world* program the statement

```
printf( "Hello World\n" ) ;
```

again consists of an expression and a terminating semi-colon where the expression here is a call to the printf standard library function.

Various expressions can be strung together to make more complicated statements but again are only terminated by a single semi-colon. For example

```
x = 2 + ( 3 * 5 ) - 23 ;
```

is a single statement that involves four different expressions.

When designing most programs we will require to build up sequences of statements. These collections of statements are called ***blocks*** and are encased between pairs of curly braces. We have already encountered these statement blocks in the case of the *main* function in the *hello world* program where the body of the function was encased in a pair of curly braces.

We will also come across statement blocks in the next few sections when we discuss some of the statements that allow us control over the execution of the simple statements.

There are two types of control statements : iteration statements that allow us to repeat one or more simple statements a certain number of times and decision statements that allow us to choose to execute one sequence of instructions over one or more others depending on certain circumstances.

Control statements are often regarded as compound statements in that they are normally combined with simpler statements which carry out the operations required. However it should be noted that each control statement is still just a single statement from the compiler's point of view.

3.2 Iteration Statements

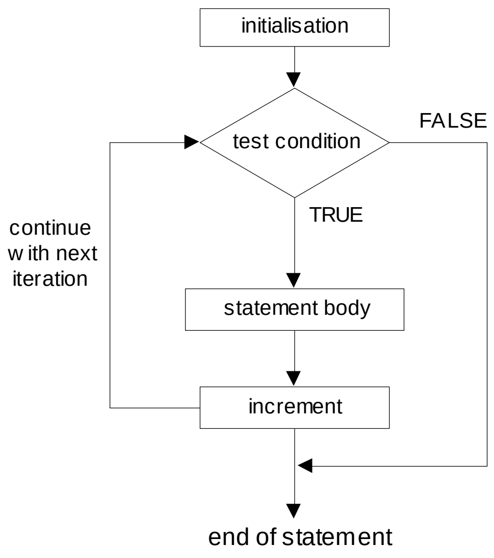
for statement

The ***for*** statement is most often used in situations where the programmer knows in advance how many times a particular set of statements are to be repeated. The for statement is sometimes termed a counted loop.

Syntax : **`for ( [initialisation] ; [condition] ; [increment] )`**
`[statement body] ;`

initialisation :- this is usually an assignment to set a loop counter variable for example.
condition :- determines when loop will terminate.
increment :- defines how the loop control variable will change each time the loop is executed.
statement body :- can be a single statement, no statement or a block of statements.

The for statement executes as follows :-



NB : The square braces above are to denote optional sections in the syntax but are not part of the syntax. The semi-colons must be present in the syntax.

For Example : Program to print out all numbers from 1 to 100.

```
#include <stdio.h>

void main()
{
  int x ;

  for ( x = 1;  x <= 100; x++ )
      printf( "%d\n", x ) ;
}
```

Curly braces are used in C to denote code blocks whether in a function as in main() or as the body of a loop.
For Example :- To print out all numbers from 1 to 100 and calculate their sum.

```
#include <stdio.h>

void main()
{
  int x, sum = 0 ;

  for ( x = 1; x <= 100; x++ )
  {
      printf( "%d\n", x ) ;
      sum += x ;
  }
  printf( "\n\nSum is %d\n", sum ) ;
}
```

Multiple Initialisations

C has a special operator called the **comma operator** which allows separate expressions to be tied together into one statement.

For example it may be tidier to initialise two variables in a for loop as follows :-

```
for ( x = 0, sum = 0; x <= 100; x++ )
      {
        printf( "%d\n", x) ;
        sum += x ;
      }
```

Any of the four sections associated with a for loop may be omitted but the semi-colons must be present always.

For Example :-

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

An infinite loop may be created as follows

```
for ( ;  ;  )
      statement body ;
```

or indeed by having a faulty terminating condition.

Sometimes a for statement may not even have a body to execute as in the following example where we just want to create a time delay.

```
for ( t = 0; t < big_num ; t++ )   ;
```

or we could rewrite the example given above as follows

```
for ( x = 1; x <= 100; printf( "%d\n", x++ ) ) ;
```

The initialisation, condition and increment sections of the for statement can contain any valid C expressions.

```
for ( x = 12 * 4 ; x < 34 / 2 * 47 ; x += 10 )
    printf( “%d “, x ) ;
```

It is possible to build a nested structure of for loops, for example the following creates a large time delay using just integer variables.

```
unsigned int x, y ;

for ( x = 0; x < 65535; x++ )
     for ( y = 0; y < 65535; y++ ) ;
```

For Example : Program to produce the following table of values

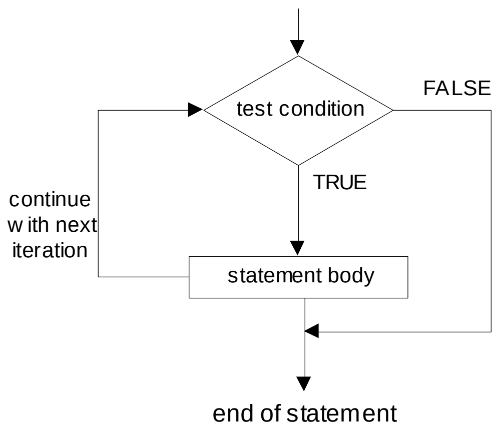
```
1  2  3  4  5
2  3  4  5  6
3  4  5  6  7
4  5  6  7  8
5  6  7  8  9
```

```
#include <stdio.h>

void main()
{
   int j, k ;

   for ( j = 1; j <= 5; j++ )
   {
     for ( k = j ; k < j + 5; k++ )
     {
        printf( "%d  ", k ) ;
     }
     printf( “\n” ) ;
   }
}
```

while statement

The ***while*** statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax : **`while ( condition )`**
`     statement body ;`



For Example : Program to sum all integers from 100 down to 1.

```
#include <stdio.h>
void main()
{
  int sum = 0, i = 100 ;

  while ( i )
      sum += i-- ;// note the use of postfix decrement operator!
  printf( "Sum is %d \n", sum ) ;
}
```

where it should be recalled that any non-zero value is deemed TRUE in the condition section of the statement.

A for loop is of course the more natural choice where the number of loop iterations is known beforehand whereas a while loop caters for unexpected situations more easily. For example if we want to continue reading input from the keyboard until the letter 'Q' is hit we might do the following.

```
char ch = '\0' ; /* initialise variable to ensure it is not 'Q'
*/

while ( ch != 'Q' )
      ch = getche() ;
```

or more succinctly

```
while ( ( ch = getche() ) != 'Q' ) ;
```

It is of course also possible to have nested while loops.

For Example : Program to guess a letter.

```
#include <stdio.h>
void main()
{
  char ch, letter = 'c'  ;       // secret letter is 'c'
  char finish = '\0' ;

  while ( finish != 'y' || finish != 'Y'  )
  {
    puts( "Guess my letter -- only 1 of 26 !" );

    while( (ch=getchar() ) != letter )// note use of parentheses
    {
     printf( "%c is wrong -- try again\n", ch ) ;
     _flushall() ;                    // purges I/O buffer
    }
    printf ( "OK you got it \n Let's start again.\n" ) ;

    letter += 3 ;// Change letter adding 3 onto ASCII value of
letter
                           //  e.g. 'c' + 3 = 'f'
    printf( "\n\nDo you want to continue (Y/N) ? ");
    finish = getchar() ;
    _flushall() ;
 }
      }
```

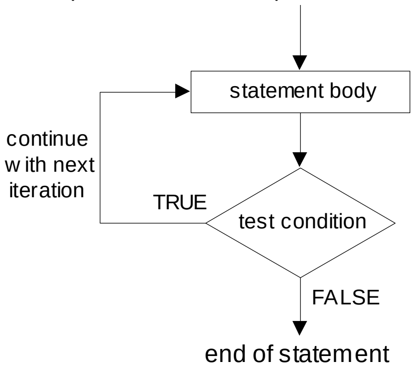
do while

The terminating condition in the for and while loops is always tested before the body of the loop is executed -- so of course the body of the loop may not be executed at all.

In the ***do while*** statement on the other hand the statement body is always executed at least once as the condition is tested at the end of the body of the loop.

Syntax :

```
do
{
statement body ;
}  while ( condition ) ;
```



For Example : To read in a number from the keyboard until a value in the range 1 to 10 is entered.

```
int i ;

do
{
  scanf( "%d\n", &i ) ;
  _flushall() ;
}   while ( i < 1 && i > 10 ) ;
```

In this case we know at least one number is required to be read so the do-while might be the natural choice over a normal while loop.

break statement

When a ***break*** statement is encountered inside a while, for, do/while or switch statement the statement is immediately terminated and execution resumes at the next statement following the statement.

For Example :-

```
           ...
for ( x = 1 ; x <= 10  ; x++  )
{
      if ( x > 4  )
            break ;

      printf( "%d  " , x ) ;
}
printf( "Next executed\n" );//Output : "1  2  3  4  Next Executed"
           ...
```

continue statement

The ***continue*** statement terminates the current iteration of a while, for or do/while statement and resumes execution back at the beginning of the loop body with the next iteration.

For Example :-

```
            ...
for ( x = 1; x <= 5; x++ )
{
      if ( x == 3 )
            continue ;
      printf( "%d  ", x ) ;
}
printf( "Finished Loop\n" ) ;    // Output : "1 2 4 5 Finished
Loop"
            ...
```

3.3 Decision Statements

if statement

The ***if*** statement is the most general method for allowing conditional execution in C.

Syntax :

```
            if  ( condition )
                  statement body ;
            else
                  statement body ;
```

or just :

```
            if ( condition )
                  statement body ;
```

In the first more general form of the statement one of two code blocks are to be executed. If the condition evaluates to TRUE the first statement body is executed otherwise for all other situations the second statement body is executed.

In the second form of the statement the statement body is executed if the condition evaluates to TRUE. No action is taken otherwise.

For Example : Program to perform integer division avoiding the division by zero case.

```
#include <stdio.h>
void main()
{
 int numerator, denominator ;

 printf( "Enter two integers as follows numerator, denominator :"
);
 scanf( "%d,  %d", &numerator, &denominator  ) ;

if ( denominator != 0 )
   printf( "%d / %d = %d \n", numerator, denominator,
                                    numerator / denominator );
else
   printf( "Invalid operation - unable to divide by zero \n" );
```

As with all other control statements the statement body can also involve multiple statements, again contained within curly braces.

Example :- Program to count the number of occurrences of the letter 'a' in an input stream of characters terminated with a carriage return.

```
#include <stdio.h>
void main()
{
  int count = 0, total = 0 ;
  char ch ;

  while (  ( ch = getchar() ) != 13 ) // 13 is ASCII value for
                                     //carriage return
  {
      if ( ch == 'a' )
      {
         count ++ ;
         printf( "\n Retrieved letter 'a' number %d\n", count ) ;
      }
      total ++ ;
      _flushall() ;
  }

  printf( "\n\n %d letters a typed in a total of %d letters.",
           count, total ) ;
      }
```

Nested if statements

if - else statements like all other decision or iteration statements in C can be nested to whatever extent is required. Care should be taken however to ensure that the if and else parts of the statement are matched correctly -- the rule to follow is that the else statement matches the most recent unmatched if statement.

For Example :-

```
if ( x > 0 )
  if ( x > 10 )
    puts ( " x is greater than zero and also greater than 10 ");
  else
    puts ("x is greater than zero but less than or equal to 10");
```

The else clause matches the most recent unmatched if clause, if (x > 10). For more clarity the above section could be rewritten as follows using curly braces with no execution penalty :-

```
if ( x > 0 )
{
 if ( x > 10 )
  puts ( " x is greater than zero and also greater than 10 ");
 else
  puts ( "x is greater than zero but less than or equal to 10 ");
}
```

if - else - if ladder

When a programming situation requires the choice of one case from many different cases successive if statements can be tied together forming what is sometimes called an if-else-if ladder.

Syntax :

```
if   ( condition_1 )
      statement_1 ;
else if ( condition_2 )
      statement_2 ;
else if ( condition_3 )
      statement_3 ;
...

else if ( condition_n )
      statement_n ;
else
      statement_default ;
```

Essentially what we have here is a complete if-else statement hanging onto each else statement working from the bottom up.

For Example : Another guessing game.

```
void main()
{
 int secret = 101, guess, count = 0 ;

 printf( "\n Try and guess my secret number.\n\n" ) ;

 while  ( 1 )          // infinite loop until we break out of it
 {
  printf( "\n Make your guess: " ) ;
  scanf( "%d", &guess ) ;
  count ++ ;

  if (   guess < secret )
     printf( "\nA little low. Try again." ) ;
  else if ( guess > secret )
     printf( "\nA little high. Try again." ) ;
  else
  {
   printf( "\nOk you got it and only on attempt %d.", count );
   break ;
   }
  }
}
```

NB : Caution is advisable when coding the if-else-if ladder as it tends to be prone to error due to mismatched if-else clauses.

Conditional Operator :- ?:

This is a special shorthand operator in C and replaces the following segment

```
if ( condition )
     expr_1 ;
else
     expr_2 ;
```

with the more elegant

```
condition ? expr_1 : expr_2 ;
```

The ?: operator is a ternary operator in that it requires three arguments. One of the advantages of the ?: operator is that it reduces simple conditions to one simple line of code which can be thrown unobtrusively into a larger section of code.

For Example :- to get the maximum of two integers, x and y, storing the larger in max.

```
max =  x >= y  ? x  :  y  ;
```

The alternative to this could be as follows

```
if ( x > = y  )
     max = x ;
else
     max = y ;
```

giving the same result but the former is a little bit more succinct.

The switch Statement

This is a multi-branch statement similar to the if - else ladder (with limitations) but clearer and easier to code.

Syntax :

```
switch ( expression )
     {
     case constant1 :   statement1 ;
                break ;

     case constant2 :   statement2 ;
                break ;

     ...

     default :   statement ;
     }
```

The value of expression is tested for equality against the values of each of the constants specified in the **case** statements in the order written until a match is found. The statements associated with that case statement are then executed until a break statement or the end of the switch statement is encountered.

When a break statement is encountered execution jumps to the statement immediately following the switch statement.

The default section is optional -- if it is not included the default is that nothing happens and execution simply falls through the end of the switch statement.

The switch statement however is limited by the following

- Can only test for equality with **integer constants** in case statements.
- No two case statement constants may be the same.
- Character constants are automatically converted to integer.

For Example :- Program to simulate a basic calculator.

```
#include <stdio.h>
void main()
{
  double num1, num2, result ;
  char op ;

  while ( 1 )
  {
    printf ( " Enter number operator number\n" ) ;
    scanf ("%f %c %f", &num1, &op, &num2 ) ;
    _flushall() ;

    switch ( op )
    {
     case '+' :  result = num1 + num2 ;
                        break ;
     case '-' :   result = num1 - num2 ;
                        break ;
     case '*' :  result = num1 * num2 ;
                        break ;
     case '/' :  if ( num2 != 0.0 ) {
                          result = num1 / num2 ;
                          break ;
                          }
     // else we allow to fall through for error message

     default :  printf ("ERROR -- Invalid operation or division
                         by 0.0" ) ;
    }
    printf( "%f %c %f = %f\n", num1, op, num2, result) ;
  }  /* while statement  */
}
```

NB : The break statement need not be included at the end of the case statement body if it is logically correct for execution to fall through to the next case statement (as in the case of division by 0.0) or to the end of the switch statement (as in the case of default :).

3.4 Efficiency Considerations

In practical programming the more elaborate algorithms may not always be the most efficient method of designing a program. In fact in many situations the simple straightforward method of solving a problem (which may be disregarded as being too simplistic) is quite often the most efficient. This is definitely true when program development time is taken into consideration but is also true in terms of the efficiency of the actual code produced.

Nevertheless quite apart from algorithmic considerations, there are a number of areas we can focus on in the code itself to improve efficiency.

One of the most important efficiency indicators is the time it takes for a program to run. The first step in eliminating sluggishness from a program is to identify which parts of the program take the most time to run and then to try and improve these areas. There are many profiling tools available which will help to establish these areas by timing a typical run of the program and displaying the time spent in each line of code and the number of times a each line of code is executed in the program. A small improvement in the efficiency of a single line of code <u>that is called many times</u> can produce dramatic overall improvements.

Most sources of inefficiency result from the inadvertent use of time expensive operations or features of the language. While modern compilers contain many advanced optimising features to make a program run faster in general the more sloppy the code the less improvements can be made by these optimisations. Therefore it is important to try and eliminate as much inefficiency as possible by adopting some simple guidelines into our programming practices.

Unnecessary Type Conversions

In many situations the fact that C is weakly typed is an advantage to the programmer but any implicit conversions allowed in a piece of code take a certain amount of time and in some cases are not needed at all if the programmer is careful. Most unnecessary conversions occur in assignments, arithmetic expressions and parameter passing.

Consider the following code segment which simply computes the sum of a user input list of integers and their average value.

```
double average, sum = 0.0 ;
short value, i ;
...
for ( i=0; i < 1000; i ++ ) {
      scanf( “%d”, &value ) ;
      sum = sum + value ;
      }
average = sum / 1000 ;
```

1. The conversion from value, of type short int, to the same type as sum, type double, occurs 1000 times in the for loop so the inherent inefficiency in that one line is repeated 1000 times which makes it substantial.

If we redefine the variable sum to be of type short we will eliminate these conversions completely. However as the range of values possible for a short are quite small we may encounter overflow problems so we might define sum to be of type long instead.

The conversion from short to long will now be implicit in the statement but it is more efficient to convert from short to long than it is from short to double.

2. Because of our modifications above the statement

```
average = sum / 1000 ;
```

now involves integer division which is not what we require here. (Note however that an implicit conversion of 1000 from int to long occurs here which may be simply avoided as follows :-

```
            average = sum / 1000L ;
```

with no time penalty whatsoever as it is carried out at compile time.)

To remedy the situation we simply do the following :-

```
            average = sum / 1000.0 ;
```

3. The statement
```
            sum = sum + value ;
```

also involves another source of inefficiency. The variable sum is loaded twice in the statement unnecessarily. If the shorthand operator += were used instead we will eliminate this.

```
            sum += value ;
```

Unnecessary Arithmetic

In general the **lowest level arithmetic** is more efficient especially in multiplication and division which are inherently expensive operations.

For Example :-
```
            double d ;
            int i ;

            d = i * 2.0 ;
```

This operation requires that the variable i is converted to double and the multiplication used is then floating point multiplication.

If we instead write the statement as

```
            d = i * 2 ;
```

we will have integer multiplication and the result is converted to double before being assigned to d. This is much more efficient than the previous and will give the same result (as long as the multiplication does not overflow).

Again very little will be saved in a single such operation but when one of many the saving may amount to something for example the expression

```
            2.0 * j * k * l * m
```

where j, k, l and m are integers might involve four floating point multiplications rather than four integer multiplications when coded with efficiency in mind.

The use of the ‘to the power of ‘ function, pow() in C, is another common example of unnecessary arithmetic.

Computing the value of num^2 or num^3 for example should never be done in a program using the pow function especially if num is an integer. This is because there is an overhead in actually calling the pow function and returning a value from it and there is an overhead if a type conversion has to be made in passing the parameters to pow() or assigning the return value from the function. Instead straightforward multiplication should be used i.e.

```
            num * num
```
rather than
```
            pow( num, 2 ) ;
```

When large powers are involved it does make sense to use the pow function but again the situation should be evaluated on its own merit.

For example if we want to print a table of num^n where n = 1 ... 99. If we do the following

```
            double num ;
            int k ;
            for ( k = 1; k <100; k++ )
              printf("%lf to %d = %lf\n", num, k, pow( num, k ));
```

we will end up with approximately $\sum n = 4950$ multiplications plus 99 function calls. Whereas if we had used
```
            double sum = num ;
            for ( k = 2; k <= 100; k++ )
                  {
                  printf( "%lf to %d = %lf\n", num, k, sum ) ;
                  sum *= num ;
                  }
```

we will require just (n -1) i.e. 98 multiplications in total.

C's Bit-wise operators may also be used to improve efficiency in certain situations.

Computing the value of 2^n can be done most efficiently using the left shift operator i.e.

```
            1 << n
```

Determining whether a value is odd or even could be done using

```
            if ( num % 2 )
                  printf( "odd" ) ;
            else
                  printf( "even" ) ;
```

but it is more efficient to use

```
            if ( num & 1 )
                  printf( "odd" ) ;
            else
                  printf( "even" ) ;
```

3.5 Exercises

1. Write a program which prints out the ASCII and hex values of all characters input at the keyboard terminating only when the character `q' or `Q' is entered.

2. Write a program to keep count of the occurrence of a user specified character in a stream of characters of known length (e.g. 50 characters) input from the keyboard. Compare this to the total number of characters input when ignoring all but alphabetic characters.

Note: The ASCII values of 'A'...'Z' are 65...90 and 'a'...'z' are 97...122.

3. Write a program to find the roots of a user specified quadratic equation.

Recall the roots of $ax^2 + bx + c = 0$ are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The user should be informed if the specified quadratic is valid or not and should be informed how many roots it has, if it has equal or approximately equal roots (b^2 == 4ac), if the roots are real
(b^2 - 4ac > 0) or if the roots are imaginary (b^2 - 4ac < 0). In the case of imaginary roots the value should be presented in the form (x + i y).

Note that C has a standard library function sqrt( ), which returns the square root of its operand, and whose prototype can be found both in the help system.

NB : The floating point number system as represented by the computer is “gappy”. Not all real numbers can be represented as there is only a limited amount of memory given towards their storage. Type float for example can only represent seven significant digits so that for example 0.1234567 and 0.1234568 are deemed consecutive floating point numbers. However in reality there are an infinite number of real numbers between these two numbers. Thus when testing for equality, e.g. testing if b^2 == 4ac, one should test for approximate equality i.e. we should test if b^2-4ac < 0.00001 where we are basically saying that accuracy to five decimal places will suffice.

4. Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard. Your program should calculate the sum and the average of the numbers input. Try and ensure that any erroneous input is refused by your program, e.g. inadvertently entering a non-numeric character etc.

5. Write a program to print out all the Fibonacci numbers using **short** integer variables until the numbers become too large to be stored in a short integer variable i.e. until overflow occurs.

a. Use a for loop construction.
b. Use a while loop construction.

Which construction is most suitable ?

Note: Fibonacci numbers are (1,1,2,3,5,8,13,...

5. Write a program which simulates the action of a simple calculator. The program should take as input two integer numbers then a character which is one of +,-,*,/,%. The numbers should be then processed according to the operator input and the result printed out. Your program should correctly intercept any possible erroneous situations such as invalid operations, integer overflow, and division by zero.

Chapter 4

Functions

As we have previously stated functions are essentially just groups of statements that are to be executed as a unit in a given order and that can be referenced by a unique name. The only way to execute these statements is by invoking them or calling them using the function's name.

Traditional program design methodology typically involves a ***top-down*** or structured approach to developing software solutions. The main task is first divided into a number simpler sub-tasks. If these sub-tasks are still too complex they are subdivided further into simpler sub-tasks, and so on until the sub-tasks become simple enough to be programmed easily.

Functions are the highest level of the building blocks given to us in C and correspond to the sub-tasks or logical units referred to above. The identification of functions in program design is an important step and will in general be a continuous process subject to modification as more becomes known about the programming problem in progress.

We have already seen many C functions such as main( ) , printf(), etc. the common trait they share being the braces that indicate they are C functions.

Syntax :
```
return_type function_name (  parameter_list )
{
body of function ;
}
```

The above is termed the **function definition** in C parlance.

Many functions will produce a result or return some information to the point at which it is called. These functions specify the type of this quantity via the ***return_type*** section of the function definition. If the return type is *void* it indicates the function returns nothing.

The ***function_name*** may be any valid C identifier and must be unique in a particular program.

If a function requires information from the point in the program from which it is called this may be passed to it by means of the ***parameter_list***. The parameter list must identify the names and types of all of the parameters to the function individually. If the function takes no parameters the braces can be left empty or use the keyword void to indicate that situation more clearly.

4.1 Function Prototype (declaration)

When writing programs in C it is normal practice to write the main() function first and to position all user functions after it or indeed in another file. Thus if a user function is called directly in main() the compiler will not know anything about it at this point i.e. if it takes parameters etc. This means we need to give the compiler this information by providing a function prototype or declaration before the function is called.

Syntax : `type_spec function_name( type_par1, type_par2, etc. );`

This declaration simply informs the compiler what type the function returns and what type and how many parameters it takes. Names may or may not be given to the parameters at this time.

For Example :- A more complicated "Hello World" program.

```
#include <stdio.h>    /* standard I/O function prototypes */

void hello( void ) ;        /* prototype  */

void main( void )
{
  hello () ;                //  function call
}

void hello ( )              // function definition
{
  printf ( "Hello World \n" ) ;
}
```

4.2 Function Definition & Local Variables

A function definition actually defines what the function does and is essentially a discrete block of code which cannot be accessed by any statement in any other function except by formally calling the function. Thus any variables declared and used in a function are private or local to that function and cannot be accessed by any other function.

For Example :-

```
#include <stdio.h>
void hello( void ) ;

void main(  )
{
hello () ;
}

void  hello ( )
{
int i ;          /*  local  or automatic variable  */

for ( i=0; i<10; i++ )
     printf( "Hello World \n" );
}
```

The variable i in the hello() function is private to the hello function i.e. it can only be accessed by code in the hello() function.

Local variables are classed as automatic variables because each time a function is called the variable is automatically created and is destroyed when the function returns control to the calling function. By created we mean that memory is set aside to store the variable's value and by destroyed we mean that the memory required is released. Thus a local variable cannot hold a value between consecutive calls to the function.

Static Local Variables

The keyword static can be used to force a local variable to retain its value between function calls.

For Example :-

```
#include <stdio.h>
void hello( void ) ;

void main ()
{
int i ;

for ( i = 0; i < 10; i++ )
     hello ( ) ;
}

void hello( )
{
static int i = 1 ;

printf( "Hello World call number %d\n", i ++ );
}
```

The static int i is created and initialised to 1 when the function is first called and only then. The variable retains its last value during subsequent calls to the function and is only destroyed when the program terminates.

NB : The variables i in main() and i in hello() are completely different variables even though they have the same name because they are private to the function in which they are declared. The compiler distinguishes between them by giving them their own unique internal names.

4.3 Scope Rules

The scope of an identifier is the area of the program in which the identifier can be accessed.

Identifiers declared inside a code block are said to have block scope. The block scope ends at the terminating } of the code block. Local variables for example are visible, i.e. can be accessed, from within the function, i.e. code block, in which they are declared. Any block can contain variable declarations be it the body of a loop statement, if statement, etc. or simply a block of code marked off by a curly brace pair. When these blocks are nested and an outer and inner block contain variables with the same name then the variable in the outer block is inaccessible until the inner block terminates.

Global variables are variables which are declared outside all functions and which are visible to all functions from that point on. These are said to have file scope.

All functions are at the same level in C i.e. cannot define a function within a function in C. Thus within the same source file all functions have file scope i.e. all functions are visible or can be called by each other (assuming they have been prototyped properly before they are called).

4.4 Returning a Value

The **return** statement is used to return a value to the calling function if necessary.

Syntax : **`return expression ;`**

If a function has a return type of type void the expression section can be omitted completely or indeed the whole return statement can be omitted and the closing curly brace of the function will cause execution to return appropriately to the calling function.

For Example :-

```
#include <stdio.h>

int hello( void ) ;

int main( )
{
int count, ch = '\0';

while (  ch  != 'q' )
      {
      count = hello( ) ;
      ch = getchar() ;
      _flushall() ;
      }

printf( "hello was called %d times\n", i ) ;
return 0 ;
}

int hello( )
{
static int i = 1 ;

printf( "Hello World \n" ) ;
// hello() keeps track of how many times it was called
return ( i++ ) ;      //  and  passes  that  information
back to its caller
}
```

NB : The return value of the function need not always be used when calling it. In the above example if we are not interested in know how often hello() has been called we simply ignore that information and invoke the function with

```
hello() ;
```

NB : When the main() function returns a value, it returns it to the operating system. Zero is commonly returned to indicate successful normal termination of a program to the operating system and other values could be used to indicate abnormal termination of the program. This value may be used in batch processing or in debugging the program.

4.5 Function Arguments

The types of all function arguments should be declared in the function prototype as well as in the function definition.

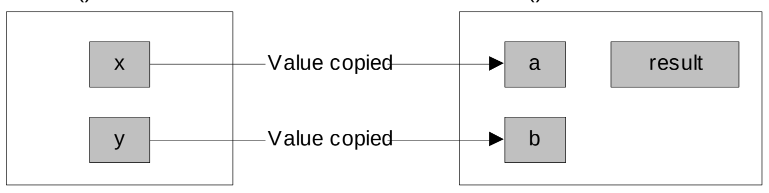
NB : In C arguments are passed to functions using the *call-by-value* scheme. This means that the compiler copies the value of the argument passed by the calling function into the formal parameter list of the called function. Thus if we change the values of the formal parameters within the called function we will have no effect on the calling arguments. The formal parameters of a function are thus local variables of the function and are created upon entry and destroyed on exit.

For Example :- Program to add two numbers.

```
     #include  <stdio.h>

     int add( int, int ) ; /*  prototype  --  need  to  indicate
types only  */

     void main ( )
     {
     int x, y ;

     puts ( "Enter two integers ") ;
     scanf( "%d %d", &x, &y) ;

     printf( "%d + %d = %d\n" , x, y, add(x,y) ) ;
     }

     int add ( int a, int b )
     {
     int result ;

     result = a + b ;
     return result ;       // parentheses used for clarity here
     }
```

NB : In the formal parameter list of a function the parameters must be individually typed.

The add() function here has three local variables, the two formal parameters and the variable result. There is no connection between the calling arguments, x and y, and the formal parameters, a and b, other than that the formal parameters are initialised with the values in the calling arguments when the function is invoked. The situation is depicted below to emphasise the independence of the various variables.



NB : The information flow is in one direction only.

For Example :- Program that attempts to swap the values of two numbers.

```
#include <stdio.h>
void swap( int, int ) ;

void main( )
{
int a, b ;

printf( "Enter two numbers" ) ;
scanf( " %d %d ", &a, &b ) ;
printf( "a = %d ;  b = %d \n", a, b ) ;

swap( a, b ) ;

printf( "a = %d ;  b = %d \n", a, b ) ;
}

void swap( int , int )//This is original form of declarator
int num1, num2 ; // which you may see in older texts and
code
{
int temp ;

temp = num2 ;
num2 = num1 ;
num1 = temp ;
}
```

Since C uses call by value to pass parameters what we have actually done in this program is to swap the values of the formal parameters but we have not changed the values in main(). Also since we can only return one value via the return statement we must find some other means to alter the values in the calling function.

The solution is to use call by reference where the addresses of the calling arguments are passed to the function parameter list and the parameters are pointers which we will encounter later on. For example when we use the scanf() standard library function to read values from the keyboard we use the & operator to give the address of the variables into which we want the values placed.

4.6 Recursion

A recursive function is a function that calls itself either directly or indirectly through another function.

Recursive function calling is often the simplest method to encode specific types of situations where the operation to be encoded can be eventually simplified into a series of more basic operations of the same type as the original complex operation.

This is especially true of certain types of mathematical functions. For example to evaluate the factorial of a number, n

$$n! = n * n-1 * n-2 * ... * 3 * 2 * 1.$$

We can simplify this operation into

$$n! = n * (n-1)!$$

where the original problem has been reduced in complexity slightly. We continue this process until we get the problem down to a task that may be solved directly, in this case as far as evaluating the factorial of 1 which is simply 1.

So a recursive function to evaluate the factorial of a number will simply keep calling itself until the argument is 1. All of the previous (n-1) recursive calls will still be active waiting until the simplest problem is solved before the more complex intermediate steps can be built back up giving the final solution.

For Example : Program to evaluate the factorial of a number using recursion.

```
#include <stdio.h>
short factorial( short ) ;
void main()
{
short i ;

printf("Enter  an  integer  and  i  will  try  to  calculate  its
factorial : " ) ;
scanf( "%d", &i ) ;
printf( "\n\nThe factorial of %d, %d! is %d\n", i, i, factorial(
i ) ) ;
}

short factorial( short num )
{
if ( num <= 1 )
      return 1 ;
else
      return ( num * factorial( num - 1 ) ) ;
}
```

This program will not work very well as is because the values of factorials grow very large very quickly. For example the value of 8! is 40320 which is too large to be held in a short so integer overflow will occur when the value entered is greater than 7. Can you offer a solution to this ?

While admittedly simple to encode in certain situations programs with recursive functions can be slower than those without because there is a time delay in actually calling the function and passing parameters to it.

There is also a memory penalty involved. If a large number of recursive calls are needed this means that there are that many functions active at that time which may exhaust the machine's memory resources. Each one of these function calls has to maintain its own set of parameters on the program stack.

4.7 #define directive

This is a preprocessor command which is used to replace any text within a C program with a more informative pseudonym.

For Example :- `#define  PI  3.14159`

When the preprocessor is called it replaces each instance of the phrase PI with the correct replacement string which is then compiled. The advantage of using this is that if we wish to change the value of PI at any stage we need only change it in this one place rather than at each point the value is used.

Macros

Macros make use of the #define directive to replace a chunk of C code to perform the same task as a function but will execute much faster since the overhead of a function call will not be involved. However the actual code involved is replaced at each call to the macro so the program will be larger than with a function.

For Example :- Macro to print an error message.

```
#define  ERROR printf( "\n Error \n" )

void main( )
{
     ...
if ( i > 1000 )
     ERROR ;         /*  note must add ; in this case to make
correct …C statement  */
}
```

Macros can also be defined so that they may take arguments.

For Example :-

```
          #define    PR( fl )  printf( "%8.2f ", fl )

          void main()
          {
          float num = 10.234 ;

          PR( num ) ;
          }
```

What the compiler actually sees is : printf("%8.2f ", num) ;

While admittedly advantageous and neat in certain situations care must be taken when coding macros as they are notorious for introducing unwanted side effects into programs.

For Example :-

```
          #define MAX(A, B)   ( ( A ) > ( B ) ? ( A ) : ( B )  )

          void main( )
          {
          int i = 20, j = 40 , k ;

          k = MAX( i++, j++ ) ;
          printf( " i = %d, j = %d\n", i, j );
          ...
          }
```

The above program might be expected to output the following

```
     i = 21, j = 41
```

whereas in fact it produces the following

```
     i = 21, j = 42
```

where the larger value is incremented twice. This is because the macro MAX is actually translated into the following by the compiler

```
        ( ( i++ ) > ( j++ ) ? ( i++ ) : ( j++ ) )
```

so that the larger parameter is incremented twice in error since parameter passing to macros is essentially text replacement.

4.8 Efficiency Considerations

As we have mentioned previously the use of functions involves an overhead in passing parameters to them and obtaining a return value from them. For this reason they can slow down your program if used excessively.

The alternative to this is to use macros in place of functions. This eliminates the penalty inherent in the function call but does make the program larger. Therefore in general macros should only be used in place of small 'would be' functions.

The penalty involved in the function call itself is also the reason why iterative methods are preferred over recursive methods in numerical programming.

4.9 Exercises

1. Write a program that can convert temperatures from the Fahrenheit scale to Celsius and back. The relationship is °C = (5/9)(°F - 32). Your program should read a temperature and which scale is used and convert it to the other, printing out the results. Write one or more functions to carry out the actual conversion.

2. Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Write functions for appropriate logical tasks in your program. You should #define all appropriate constants in your program.

3. Write and test a function to convert an unsigned integer to binary notation. Use the sizeof operator to make the program machine independent i.e. portable.

4. Write and test two functions, one that packs two character variables into one integer variable and another which unpacks them again to check the result.

5. Write and test a circular shift left function for one byte unsigned variables i.e. unsigned characters.

e.g. 10011100 circular shift left by 2 yields 01110010.

6. An integer is said to be *prime* if it is divisible only by one and itself. Write a function which determines whether an integer is *prime* or not. To test your function write a program that prints out all prime numbers between 1 and 10,000.

7. Write and test a function to read in a signed integer from the standard input device. Your function should not be allowed to use the scanf function. The function should have the prototype

```
int getint( void ) ;
```

and should be able to accommodate the presence of a minus sign, a plus sign or no sign (i.e. positive by default). Note that the ASCII values of the digits 0 - 9 are consecutive, 48 - 57 and that if we wish to convert the digit '3' for example to an integer we simply subtract the ASCII value of digit '0' from its ASCII value i.e. '3' - '0' = 51 - 48 = 3.

8. Write and test a function to get a floating point number from the standard input device. The function should have the prototype

```
float getfloat( void ) ;
```

and should be able to accommodate normal floating point notation and exponential notation. You should make use of the `getint()` function in exercise 5. to read in the various components of the floating point number.

9. Write and test a function

```
double  power ( double x,  int n );
```

to calculate the value of x raised to the power of n.

(a) Use your own calculations.

(b) Use the standard library function `pow()` (for more information use help system) paying particular attention to type compatibilities.

10. Write a recursive function to calculate and print out all the Fibonacci values up to and including the nth . Recall that the Fibonacci series is 1, 1, 2, 3, 5, 8, 13, ...To test your program allow the user to enter the value of n and then print out the series. The program should run continuously until it is explicitly terminated by the user.

11. Programming Assignment : Non-linear Equations.

Your basic task in this assignment is to write a C program which will allow the user to solve the following non-linear equation

$$f(x) = \sin \frac{2x}{5} - x + 1 = 0$$

to an accuracy of 10^{-5} using both the Bisection method and Newton's method.

In order to be able to solve equations using these methods you will need to provide your program with suitable initial estimates to the actual root, an interval over which the sign of f(x) changes in the case of the Bisection method and an initial estimate x_0 where f'(x_0) is not very small in the case of Newton's method.

To help the user to provide such estimates your program should first tabulate the function f(x), repeatedly until the user is satisfied, over a user specified interval [x_1, x_2] and using a user specified tabulation step.

This should allow the user to begin by tabulating the function over a large interval with a coarse tabulation step and to fine down the interval and step until he / she can determine the behaviour of f(x) in or about the actual root.

Once the user is satisfied with the tabulation your program should read in the appropriate initial estimates from the user and test their validity and then solve the equation using both methods. Finally your program should compare the number of iterations required to compute the root using both methods and print out the value of the root.

You should break your program up into appropriate **logical** functions and try to intercept all erroneous situations as soon as possible and take appropriate action.

Note : The exact solution to the above equation is 1.595869359.

Chapter 5

Arrays & Strings

An array is a collection of variables of the same type that are referenced by a common name. Specific elements or variables in the array are accessed by means of an index into the array.

In C all arrays consist of contiguous memory locations. The lowest address corresponds to the first element in the array while the largest address corresponds to the last element in the array.

C supports both single and multi-dimensional arrays.

5.1 Single Dimension Arrays

Syntax : **type var_name[size] ;**

where type is the type of each element in the array, var_name is any valid C identifier, and size is the number of elements in the array which has to be a constant value.

NB : In C all arrays use zero as the index to the first element in the array.

For Example :-

```
            int array[ 5 ] ;
```

which we might illustrate as follows for a 32-bit system where each int requires 4 bytes.

array[0]	12	loc^n 1000
array[1]	-345	loc^n 1004
array[2]	342	loc^n 1008
array[3]	-30000	loc^n 1012
array[4]	23455	loc^n 1016

NB : The valid indices for array above are 0 .. 4, i.e. 0 .. number of elements - 1

For Example :- To load an array with values 0 .. 99

```
            int x[100] ;
            int i ;

            for ( i = 0; i < 100; i++ )
                  x[i] = i ;
```

Arrays should be viewed as just collections of variables so we can treat the individual elements in the same way as any other variables. For example we can obtain the address of each one as follows to read values into the array

```
            for ( i = 0; i < 100; i++ ) {
                  printf( "Enter element %d", i + 1 ) ;
                  scanf( "%d\n", &x[i] ) ;
                  }
```

NB : Note the use of the printf statement here. As arrays are normally viewed as starting with index 1 the user will feel happier using this so it is good policy to use it in "public".

To determine to size of an array at run time the sizeof operator is used. This returns the size in bytes of its argument. The name of the array is given as the operand

```
            size_of_array = sizeof ( array_name )  ;
```

NB : C carries out no boundary checking on array access so the programmer has to ensure he/she is within the bounds of the array when accessing it. If the program tries to access an array element outside of the bounds of the array C will try and accommodate the operation. For example if a program tries to access element array[5] above which does not exist the system will give access to the location where element array[5] should be i.e. 5 x 4 bytes from the beginning of the array.

array[0]	12	loc^n 1000
array[1]	-345	loc^n 1004
array[2]	342	loc^n 1008
array[3]	-30000	loc^n 1012
array[4]	23455	loc^n 1016
array[5]	123	loc^n 1020

This piece of memory does not belong to the array and is likely to be in use by some other variable in the program. If we are just reading a value from this location the situation isn't so drastic our logic just goes haywire. However if we are writing to this memory location we will be changing values belonging to another section of the program which can be catastrophic.

Initialising Arrays

Arrays can be initialised at time of declaration in the following manner.

```
      type array[ size ] = { value list };
```

For Example :-
```
            int i[5] = {1, 2, 3, 4, 5 } ;
```

```
i[0] = 1, i[1] = 2, etc.
```

The size specification in the declaration may be omitted which causes the compiler to count the number of elements in the value list and allocate appropriate storage.

For Example :- `int i[ ]  =  { 1, 2, 3, 4, 5 } ;`

5.2 Strings

In C a string is defined as a character array which is terminated by a special character, the null character '\0', as there is no string type as such in C.

Thus the string or character array must always be defined to be one character longer than is needed in order to cater for the '\0'.

For Example :- string to hold 5 characters

```
char s[6] ;
```

					'\0'

A string constant is simply a list of characters within double quotes e.g. "Hello" with the '\0' character being automatically appended at the end by the compiler.

A string may be initialised as simply as follows

```
char s[6] = "Hello" ;
```

'H'	'e'	'l'	'l'	'o'	'\0'

as opposed to

```
        char s[6] = { 'H', 'e', 'l', 'l', 'o', '\0' } ;
```

Again the size specification may be omitted allowing the compiler to determine the size required.

Manipulating Strings

We can print out the contents of a string using printf() as we have seen already or by using puts().

```
printf( "%s", s ) ;
puts( s ) ;
```

Strings can be read in using scanf()

```
scanf( "%s", s ) ;
```

where we do not require the familiar & as **the name of an array without any index or square braces is also the address of the array**.

A string can also be read in using gets()

```
gets ( s ) ;
```

There is also a wide range of string manipulation functions included in the C Standard Library which are prototyped in <string.h> which you should familiarise yourself with.

For Example :-

```
char s1[20] = “String1”,  s2[20] = “String2” ;
int i ;

strcpy( s1, s2 )  ;   /*   copies s2 into s1. */

i = strcmp( s1,s2 ) ; /*  compares s1 and s2. It returns zero if
             s1 same as s2,-1 if s1 < s2, and +1 if s1 > s2 */

i = strlen( s1 ) ;          /* returns the length of s1 */

strcat ( s1, s2 ) ;         /* Concatenates s2 onto end of s1  */
```

5.3 Multidimensional Arrays

Multidimensional arrays of any dimension are possible in C but in practice only two or three dimensional arrays are workable. The most common multidimensional array is a two dimensional array for example the computer display, board games, a mathematical matrix etc.

Syntax : **`type name [ rows ] [ columns ] ;`**

For Example :- 2D array of dimension 2 X 3.

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

d[0][0]	d[0][1]	d[0][2]
d[1][0]	d[1][1]	d[1][2]

A two dimensional array is actually an array of arrays, in the above case an array of two integer arrays (the rows) each with three elements, and is stored row-wise in memory.

For Example :- Program to fill in a 2D array with numbers 1 to 6 and to print it out row-wise.

```
#include <stdio.h>
void main( )
{
int i, j, num[2][3] ;

for ( i = 0; i < 2; i++ )
      for ( j = 0; j < 3; j ++ )
            num[i][j] =  i * 3 + j + 1 ;

for ( i = 0; i < 2; i++ )
      {
      for ( j = 0; j < 3; j ++ )
            printf("%d ",num[i][j]  ) ;
      printf("\n" );
      }
}
```

For Example :- Program to tabulate ***sin(x)*** from x = 0 to 10 radians in steps of 0.1 radians.

```
#include <stdio.h>
#include <math.h>
void main()
{
int i  ;
double x ;
double table[100][2] ;// we will need 100 data points for
                      // the above range and step size and
                      // will store both x and f(x)
for ( x = 0.0, i = 0; x < 10.0; x += 0.1, i++ ) {
  table[i][0] = x ;
  table[i][1] = sin( x ) ;
  printf("\n Sin( %lf ) = %lf", table[i][0], table[i][1] );
}
}
```

To initialise a multidimensional array all but the leftmost index must be specified so that the compiler can index the array properly.

For Example :-

```
int d[ ] [ 3 ] = { 1, 2, 3, 4, 5, 6 } ;
```

However it is more useful to enclose the individual row values in curly braces for clarity as follows.

```
int d[ ] [ 3 ] = { {1, 2, 3}, {4, 5, 6} } ;
```

5.4 Arrays of Strings

An array of strings is in fact a two dimensional array of characters but it is more useful to view this as an array of individual single dimension character arrays or strings.

For Example :-

```
char str_array[ 10 ] [ 30 ] ;
```

where the row index is used to access the individual row strings and where the column index is the size of each string, thus str_array is an array of 10 strings each with a maximum size of 29 characters leaving one extra for the terminating null character.

For Example :- Program to read strings into str_array and print them out character by character.

```
#include <stdio.h>

char str_array[10][30] ;

void main()
{
int i, j ;

puts("Enter ten strings\n") ;

for ( i = 0 ; i < 10; i++ ) // read in as strings so a single for
                            // loop suffices
{
printf( " %d : ", i + 1) ;
gets( str_array[i] ) ;
}

for ( i = 0; i < 10; i++ )//printed out as individual chars so a
{                         // nested for loop structure is required
  for ( j=0; str_array[i][j] != '\0' ; j++ )
     putchar ( str_array[i][j] ) ;
  putchar( '\n' ) ;
}
}
```

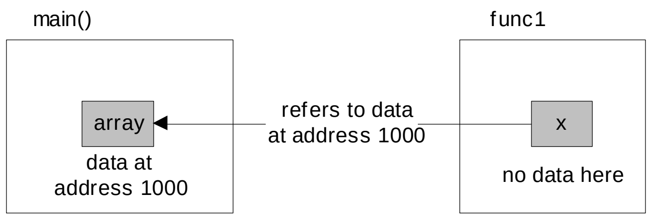
5.5 Arrays as arguments to functions (1D)

In C it is impossible to pass an entire array as an argument to a function -- instead the address of the array is passed as a parameter to the function. (In time we will regard this as a pointer).

The name of an array without any index is the address of the first element of the array and hence of the whole array as it is stored contiguously. However we need to know the size of the array in the function - either by passing an extra parameter or by using the sizeof operator..

For Example :-
```
void main()
{
  int array[20] ;

  func1( array ) ;/* passes pointer to array to func1 */
}
```

Since we are passing the address of the array the function will be able to manipulate the actual data of the array in main(). This is call by reference as we are not making a copy of the data but are instead passing its address to the function. Thus the called function is manipulating the same data space as the calling function.



In the function receiving the array the formal parameters can be declared in one of three almost equivalent ways as follows :-

- As a sized array :
```
          func1 ( int x[10] ) {
          ...
          }
```
- As an unsized array :
```
          func1 ( int x[ ] ) {
          ...
          }
```
- As an actual pointer
```
          func1 ( int *x ) {
          ...
          }
```

All three methods are identical because each tells us that in this case the address of an array of integers is to be expected.

Note however that in cases 2 and 3 above where we specify the formal parameter as an unsized array or simply as a pointer we cannot determine the size of the array passed in using the sizeof operator as the compiler does not know what dimensions the array has at this point. Instead sizeof returns the size of the pointer itself, two in the case of near pointers in a 16-bit system but four in 32-bit systems.
For Example :- Program to calculate the average value of an array of doubles.

```
#include <stdio.h>
void read_array( double array[ ], int size ) ;
double mean( double array[ ], int size ) ;

void main()
{
double data[ 100 ] ;
double average ;

read_array( data, 100 ) ;
average = mean( data, 100 ) ;
}

void read_array( double array[ ], int size )
{
int i ;

for ( i = 0; i<100; i++ )   {
     printf( "\nEnter data value %d : i + 1 );
     scanf( "%lf", &array[i] ;
     _flushall() ;
     }
}

double mean( double array[ ], int size )
{
double total = 0.0 ;
int count = size ;
while ( count-- )  // size is a local variable which we can
                   // use at will
     total += array[ count ] ;
return ( total / size ) ;
}
```

For Example :- Program to test if a user input string is a palindrome or not.

```
#include  <stdio.h>
int palin( char array[ ] ) ;    /* Function to determine if array
  is a palindrome returns 1 if it is a palindrome, 0 otherwise */
void main( )
{
char str[100] ;

puts( "Enter test string" ) ;
gets( str ) ;

if ( palin( str ) )
    printf( "%s is a palindrome\n", str ) ;
else
    printf( "%s is not a palindrome\n") ;
}
```

```
      int palin ( char array[ ] )
      {
      int i = 0, j = 0 ;

      while ( array[j++] ) ;      /*  get  length  of  string   i.e.
increment j while array[j] != '\0' */
      j -=  2 ;              /*  move  back  two  --  gone  one  beyond
'\0' */

      for (  ; i < j ; i++, j-- )
           if ( array[ i ] != array[ j ]
                return 0 ;            /*  return  value  0  if  not  a
palindrome */

      return 1 ;             /* otherwise it is a palindrome */
      }
```

An alternative way of writing the palin() function might be as follows using string manipulation functions (must add #include <string.h> to top of file in this case).

```
      int palin( char array[ ] )
      {
      char temp[30] ;

      strcpy( temp, array ) ;/* make a working copy of string */

      strrev( temp ) ;       /* reverse string */

      if ( ! strcmp( temp, array ) )   /* compare strings –
                                     if same strcmp returns 0  */
           return 1 ;
      else
           return 0 ;
      }
```

5.6 Passing Multidimensional Arrays

Function calls with multi-dimensional arrays will be the same as with single dimension arrays as we will still only pass the address of the first element of the array.

However to declare the formal parameters to the function we need to specify all but one of the dimensions of the array so that it may be indexed properly in the function.

For Example :-

2D array of doubles :-	`double x[10][20] ;`
Call func1 with x a parameter :-	`func1( x ) ;`
Declaration in func1 :-	`func1( double y[ ][20] ) { ... }`

The compiler must at least be informed how many columns the matrix has to index it correctly. For example to access element y[5][3] of the array in memory the compiler might do the following

```
      element No =  5 * 20 + 3 = 103.
```

NB : Multi-dimensional arrays are stored row-wise so `y[5][3]` is the 4th element in the 6th row.

Since we are dealing with an array of doubles this means it must access the memory location 103 X 8 bytes from the beginning of the array.

Thus the compiler needs to know how many elements are in each row of the 2D array above. In general the compiler needs to know all dimensions except the leftmost at the very least.

For Example :- Program to add two 2 x 2 matrices.

```
#include < stdio.h>

void mat_read( int mat[2][2] ) ; // Write these two functions for
                                 //yourselves
void mat_print( int mat[2][2] ) ;

void mat_add( int mat1[ ][2], int mat2[ ][2], int mat3[ ][2] ) ;

void main()
{
int mat_a[2][2], mat_b[2][2], mat_res[2][2] ;

puts( "Enter Matrix a row-wise :-\n" );
mat_read( mat_a ) ;
puts( "\nMatrix a is :-\n" ) ;
mat_print( mat_a ) ;
puts( "Enter Matrix b row-wise" );
mat_read( mat_b ) ;
puts( "\nMatrix b is :-\n" ) ;
mat_print( mat_b ) ;

mat_add( mat_a, mat_b, mat_res ) ;

puts( "The resultant matrix is\n" ) ;
mat_print( mat_res ) ;
}

void mat_add( int mat1[ ][2], int mat2[ ][2], int mat3[ ][2] )
{
int j, k ;

for ( j = 0; j < 2; j++ )
      for ( k = 0; k < 2; k++ )
            mat_res[j][k] = mat1[j][k] + mat2[j][k]   ;
}
```

5.7 Exercises

1. Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard, storing them in an array of maximum size 100 say. Your program should then calculate the sum and the average of the numbers input.

2. Modify your program in exercise 1 so that the maximum and minimum values in the data are found and are ignored when calculating the average value.

3. Write a program that allows the elements of a user input array of doubles to be reversed so that first becomes last etc. Use a separate swap function to carry out the switching of elements.

4. Write a program that reads an array of up to 20 integers from the keyboard and produces a histogram of the values as indicated below.

```
|          *
|        * *
|        * *
|      * * * *
|  *   * * * *   *
|  * * * * * *   *
|* * * * * * * * *
|1 3 2 4 6 7 4 1 3
```

A two dimensional array of characters should be used to produce the histogram, filling it with asterisks appropriate to the data values and then just printing out the array. Some scaling will be required if the values are allowed to exceed the number of asterisks that can fit on the screen vertically.

5. Write a program to accept two strings from the keyboard, compare them and then print out whether or not they are the same.

6(a). Write a function to test whether or not a word is a palindrome e.g. MADAM.

(b). Modify the function in 2(a) so that white space characters are ignored i.e. so that MADAM IM ADAM for example is deemed a palindrome.

7. Write and test a function that inserts a character anywhere in a string. The function should take the general form

```
strins(  char *string,  char character,  int position )
```

Notes :-

1. Recall Microsoft C supports a broad range of string handling functions such as strlen(), strcpy(), etc.

2. Your function should provide sufficient error checking to prevent erroneous operation e.g. your function should check that the desired position actually exists.

3. Your function need not consider memory / space requirements placing the onus on the user to make sure sufficient space is available in the operands.

Modify the strins( ) function written above so that it allows a string of characters rather than an individual character to be inserted at the designated position.

8. Write a program that multiplies two 2 X 2 matrices and prints out the resultant matrix. The program should read in the individual matrices and display them in standard format for the user to check them. The user should be allowed to correct/alter any one element of the matrix at a time and check the revised matrix until satisfied. The results of the operation should be displayed along with the two component matrices.

Recall that

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} . \begin{pmatrix} c_1 & d_1 \\ c_2 & d_2 \end{pmatrix} = \begin{pmatrix} a_1c_1 + b_1c_2 & a_1d_1 + b_1d_2 \\ a_2c_1 + b_2c_2 & a_2d_1 + b_2d_2 \end{pmatrix}$$

9. A real polynomial p(x) of degree n is given by

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

where the coefficients a_n are real numbers.

Write a function

```
double  eval ( double p[], double x,  int n );
```

which calculates the value of a polynomial of degree n at a value x using the coefficients in the array p[]. Your program should read in the values of the coefficients a_n and store them in an array and present the results of the calculation.

(a) Use straightforward calculations to compute the value.
(b) Employ Horner's Rule to calculate the value.

Recall Horner's Rule for a three degree polynomial for example is

$$p(x) = a0 + x(a_1 + x(a_2 + x(a_3)))$$

Compare the efficiency obtained using both methods.

10. Programming Assignment : Tic-Tac-Toe

Write a C program which allows the user to play the computer in a game of tic--tac--toe (noughts and crosses).

The program should use a two dimensional array to represent the 3x3 board matrix and allow the various positions on the board to be referenced as co-ordinates such as (0,0) for the upper--left corner and (2,2) for the lower--right corner.

Your program might include functions to do the following :-

1. Initialise the playing area appropriately.
2. Display the current state of the game.
3. Get the player's move.
4. Get the computer's move.
5. Check if there is a winner.

The character 'O' should be used to indicate a computer occupied position and the character 'X' to indicate that of a player. The space character might be used to indicate an unoccupied location.

In displaying the current state of the board simple formatted text output will suffice (clearing the screen between move updates etc.) as opposed to providing DOS graphics support.

The tic tac toe program should incorporate the following three user specified options in a single package.

<u>Option 1:</u> The computer will be limited to playing a very simple dull game. When it is the computer's turn to move it simply scans the board matrix and fills the first unoccupied position encountered. If it cannot find an unoccupied location it reports a drawn game.

<u>Option 2:</u> The computer plays a full game ie. it checks first if it can win the game with its current move. This failing it checks if it can prevent the player from winning with his/her next move. Otherwise it fills the first available unoccupied position.

<u>Option 3:</u> The program allows two players to play each other (without any help or hints!).

The program should run continuously allowing the player to exit using a special "hot-key" during a game and should allow the player to choose who moves first.

Chapter 6

Pointers

Pointers are without a doubt one of the most important mechanisms in C. They are the means by which we implement *call by reference* function calls, they are closely related to arrays and strings in C, they are fundamental to utilising C's dynamic memory allocation features, and they can lead to faster and more efficient code when used correctly.

A ***pointer*** is a ***variable*** that is used to store a ***memory address***. Most commonly the address is the location of another variable in memory.

If one variable holds the address of another then it is said to point to the second variable.

Address	Value	Variable
1000		
1004	1012	ivar_ptr
1008		
1012	23	ivar
1016		

In the above illustration *ivar* is a variable of type int with a value 23 and stored at memory location 1012. *ivar_ptr* is a variable of type *pointer to int* which has a value of 1012 and is stored at memory location 1004. Thus ivar_ptr is said to point to the variable ivar and allows us to refer indirectly to it in memory.

NB : It should be remembered that ivar_ptr is a variable itself with a specific piece of memory associated with it, in this 32-bit case four bytes at address 1004 which is used to store an address.

6.1 Pointer Variables

Pointers like all other variables in C must be declared as such prior to use.

Syntax : `type *ptr ;`

which indicates that *ptr* is a pointer to a variable of type *type*. For example

```
int  *ptr ;
```

declares a pointer `ptr` to variables of type `int`.

NB : The type of the pointer variable ptr is ***`int *`***. The declaration of a pointer variable normally sets aside just two or four bytes of storage for the pointer whatever it is defined to point to.

In 16-bit systems two byte pointers are termed near pointers and are used in small memory model programs where all addresses are just segment offset addresses and 16 bits in length. In larger memory model programs addresses include segment and offset addresses and are 32 bits long and thus pointers are 4 bytes in size and are termed far pointers.

In 32-bit systems we have a flat address system where every part of memory is accessible using 32-bit pointers.

6.2 Pointer Operators * and &

& is a unary operator that returns the **address** of its operand which must be a variable.

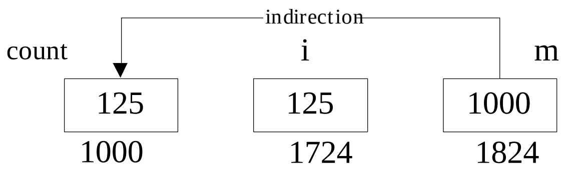
For Example :-
```
int *m ;
int count=125, i ;/* m is a pointer to int, count, i are integers
*/
m = &count ;
```

The address of the variable count is placed in the pointer variable m.

The * operator is the complement of the address operator & and is normally termed the indirection operator. Like the & operator it is a unary operator and it returns the **value** of the variable located at the address its operand stores.

For Example :-
```
            i = *m ;
```

assigns the value which is located at the memory location whose address is stored in m, to the integer i. So essentially in this case we have assigned the value of the variable count to the variable i. The final situation is illustrated below.



One of the most frequent causes of error when dealing with pointers is using an uninitialised pointer. Pointers should be initialised when they are declared or in an assignment statement. Like any variable if you do not specifically assign a value to a pointer variable it may contain any value. This is extremely dangerous when dealing with pointers because the pointer may point to any arbitrary location in memory, possibly to an unused location but also possibly to a memory location that is used by the operating system. If your program tries to change the value at this address it may cause the whole system to crash. Therefore it is important to initialise all pointers before use either explicitly in your program or when defining the pointer.

A pointer may also be initialised to 0 (zero) or NULL which means it is pointing at nothing. This will cause a run-time error if the pointer is inadvertently used in this state. It is useful to be able to test if a pointer has a null value or not as a means of determining if it is pointing at something useful in a program.

NB : NULL is #defined in <stdio.h>.

For Example :-
```
            int var1, var2 ;
            int *ptr1, *ptr2 = &var2 ;
            int *ptr3 = NULL ;
            ...
            ptr1 = &var1 ;
```

`ptr1` and `ptr2` are now pointing to data locations within the program so we are free to manipulate them at will i.e. we are free to manipulate the piece of memory they point to.

6.3 Call by Reference

Recall when we wanted to swap two values using a function we were unable to actually swap the calling parameters as the call by value standard was employed. The solution to the problem is to use call by reference which is implemented in C by using pointers as is illustrated in the following example.

```
#include <stdio.h>

void swap( int *, int  * ) ;

void main( )
{
int a, b ;

printf( "Enter two numbers" ) ;
scanf( " %d %d ", &a, &b ) ;
printf( "a = %d ;  b = %d \n", a, b ) ;

swap( &a, &b ) ;

printf( "a = %d ;  b = %d \n", a, b ) ;
}
void swap ( int  *ptr1, int  *ptr2 )
{
int temp ;

temp = *ptr2 ;
*ptr2 = *ptr1 ;
*ptr1 = temp ;
}
```

The `swap()` function is now written to take integer pointers as parameters and so is called in `main()` as

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

where the addresses of the variables are passed and copied into the pointer variables in the parameter list of swap(). These pointers must be de-referenced to manipulate the values, and it is values in the the same memory locations as in main() we are swapping unlike the previous version of swap where we were only swapping local data values.

In our earlier call-by-value version of the program we called the function from main() as *`swap(a,b);`* and the values of these two calling arguments were copied into the formal arguments of function swap.

In our call-by-reference version above our formal arguments are pointers to int and it is the addresses contained in these pointers, (i.e. the pointer values), that are copied here into the formal arguments of the function. However when we de-reference these pointers we are accessing the values in the main() function as their addresses do not change.

6.4 Pointers and Arrays

There is a very close relationship between pointer and array notation in C. As we have seen already the name of an array (or string) is actually the address in memory of the array and so it is essentially a constant pointer.

For Example :-
```
char str[80], *ptr ;

ptr =  str ;/* causes ptr to point to start of string str  */
ptr = &str[0] ;  /* this performs the same as above */
```

It is illegal however to do the following

```
str = ptr ;      /* illegal */
```

as str is a constant pointer and so its value i.e. the address it holds cannot be changed.

Instead of using the normal method of accessing array elements using an index we can use pointers in much the same way to access them as follows.

```
char str[80], *ptr , ch;

ptr = str ;           // position the pointer appropriately

*ptr = 'a' ;          // access first element i.e. str[0]
ch = *( ptr + 1 )  ;  // access second element i.e. str[1]
```

Thus ***(array + index)** is equivalent to **array[index]**.

Note that the parentheses are necessary above as the precedence of * is higher than that of +. The expression
```
            ch = *ptr + 1 ;
```

for example says to access the character pointed to by ptr (str[0] in above example with value ‘a’) and to add the value 1 to it. This causes the ASCII value of ‘a’ to be incremented by 1 so that the value assigned to the variable ch is ‘b’.

In fact so close is the relationship between the two forms that we can do the following

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

ptr = x ;
ptr[4] = 10 ;   /* accesses element 5 of array by indexing a pointer */
```

6.5 Pointer Arithmetic

Pointer variables can be manipulated in certain limited ways. Many of the manipulations are most useful when dealing with arrays which are stored in contiguous memory locations. Knowing the layout of memory enables us to traverse it using a pointer and not get completely lost.

- **Assignment**

```
int count, *p1, *p2 ;

p1 = &count ;         //  assign  the  address  of  a  variable
directly
p2 = p1 ;       // assign the value of another pointer variable,
an address
```

- **Addition / Subtraction**

The value a pointer holds is just the address of a variable in memory, which is normally a four byte entity. It is possible to modify this address by integer addition and subtraction if necessary. Consider the following we assume a 32-bit system and hence 32-bit integers.

```
int *ptr ;
int array[3] = { 100, 101,
102 } ;
ptr = array ;
```

	Address	Value
ptr	1000	2008
	①	①
array[0]	2008	100
array[1]	2012	101
array[2]	2016	102

We now have the pointer variable ptr pointing at the start of *array* which is stored at memory location 2008 in our illustration. Since we know that element array[1] is stored at address 2012 directly after element array[0] we could perform the following to access its value using the pointer.

```
ptr += 1 ;
```

This surprisingly will cause ptr to hold the value 1012 which is the address of array[1], so we can access the value of element array[1]. The reason for this is that ptr is defined to be a pointer to type int, which are four bytes in size on a 32-bit system. When we add 1 to ptr what we want to happen is to point to the **next integer** in memory. Since an integer requires four bytes of storage the compiler increments ptr by 4. Likewise a pointer to type char would be incremented by 1, a pointer to float by 4, etc.

Similarly we can carry out integer subtraction to move the pointer backwards in memory.

```
ptr = ptr - 1 ;
ptr -= 10 ;
```

The shorthand operators ++ and -- can also be used with pointers. In our continuing example with integers the statement ***ptr++ ;*** will cause the address in ptr to be incremented by 4 and so point to the next integer in memory and similarly ***ptr-- ;*** will cause the address in ptr to be decremented by 4 and point to the previous integer in memory.

NB : Two pointer variables may not be added together (it does not make any logical sense).

```
char *p1, *p2 ;
p1 = p1 + p2 ;   /* illegal operation */
```

Two pointers may however be subtracted as follows.

```
int *p1, *p2, array[3], count ;
p1 = array ;
p2 = &array[2] ;
```

```
            count = p2 - p1 ;      /* legal */
```

The result of such an operation is not however a pointer, it is the number of elements of the base type of the pointer that lie between the two pointers in memory.

- **Comparisons**

We can compare pointers using the relational operators ==, <, and > to establish whether two pointers point to the same location, to a lower location in memory, or to a higher location in memory. These operations are again used in conjunction with arrays when dealing with sorting algorithms etc.

For Example :- Writing our own version of the puts() standard library function.

1. Using array notation

```
void puts( const char s[ ] ) /* const keyword makes string
                                contents read only */
{
int i ;

for ( i = 0; s[i] ; i++ )
      putchar( s[i] ) ;
putchar( '\n' ) ;
}
```

2. Using pointer notation

```
void puts( const char *s ) // char *const s would make pointer
unalterable
{
 while ( *s )
     putchar( *s++ ) ;
 putchar( '\n' ) ;
}
```

As you can see by comparing the two versions above the second version using pointers is a much simpler version of the function. No extra variables are required and it is more efficient as we will see because of its use of pointer indirection.

For Example :- Palindrome program using pointers.

```
#include  <stdio.h>
int palin( char * ) ;      /* Function to determine if array is a palindrome. returns 1 if
                                         it is a palindrome, 0 otherwise */
void main( )
{
char str[30], c ;

puts( "Enter test string" ) ;
gets( str ) ;

if ( palin( str ) )
        printf( "%s is a palindrome\n", str ) ;
else
        printf( "%s is not a palindrome\n") ;
}
```

```
int palin ( char *str )
{
char *ptr ;

ptr = str ;
while ( *ptr  )
        ptr++ ;             /* get length of string  i.e. increment ptr while *ptr != '\0' */
ptr-- ;                     /* move back one from '\0'                            */

while ( str < ptr )
        if ( *str++  !=  *ptr-- )
                return 0 ;                      /* return value 0 if not a palindrome */

return 1 ;                  /* otherwise it is a palindrome */
}
```

Strings and pointers

C's standard library string handling functions use pointers to manipulate the strings. For example the prototype for the strcmp() function found in <string.h> is

```
int strcmp( const char *string1, const char *string2 ) ;
```

where const is a C keyword which locks the variable it is associated with and prevents any inadvertent changes to it within the function.

Strings can be initialised using pointer or array notation as follows

```
char *str = "Hello\n" ;
char string[] = "Hello\n" ;
```

in both cases the compiler allocates just sufficient storage for both strings.

6.6 Arrays of Pointers

It is possible to declare arrays of pointers in C the same as any other 'type'. For example

```
int *x[10] ;
```

declares an array of ten integer pointers.

To make one of the pointers point to a variable one might do the following.

```
x[ 2 ]  =  &var ;
```

To access the value pointed to by x[2] we would do the following

```
*x[ 2 ]
```

which simply de-references the pointer x[2] using the * operator.

Passing this array to a function can be done by treating it the same as a normal array which happens to be an array of elements of type int *.

For Example : -

```
        void display( int *q[ ], int size )
        {
        int t ;
        for ( t=0; t < size; t++ )
                printf( "%d ", *q[t] ) ;
        }
```

Note that q is actually a pointer to an array of pointers as we will see later on with multiple indirection.

A common use of pointer arrays is to hold arrays of strings.

For Example :- A function to print error messages.

```
        void serror( int num )
        {
        static char *err[] = {
                "Cannot Open File\n",
                "Read Error\n",
                "Write Error\n" } ;

        puts( err[num] );
        }
```

Note that using an array of pointers to char initialised as above conserves space as no blank filling characters are required as would be if we used

```
    char err[3][30] = {
        ... } ;
```

6.7 Command Line Arguments

Command line arguments allow us to pass information into the program as it is run. For example the simple operating system command *type* uses command line arguments as follows

```
    c:>type text.dat
```

where the name of the file to be printed is taken into the type program and the contents of the file then printed out.

In C there are two in-built arguments to the main() function commonly called **argc** and **argv** which are used to process command line arguments.

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

argc is used to hold the total number of arguments used on the command line which is always at least one because the program name is considered the first command line argument.
argv is a pointer to an array of pointers to strings where each element in argv points to a command line argument. For example argv[0] points to the first string, the program name.

For Example :- Program to print a name (saved in name.c) using command line arguments.

```
#include <stdio.h>
void main( int argc, char *argv[ ] )
{
if ( argc != 2 )
        {
        puts( "Missing parameter.  Usage : name yourname" ) ;
        exit( 1 );
        }
printf( "Hello %s", argv[1] ) ;
 }
```

To run the program one might type

```
c:\>name tom
```

For Example :- Program to count down from a given value, the countdown being displayed if the argument "display" is given.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

void main( int argc, char *argv[ ] )
{
int disp, count ;

if ( argc < 2 )
        {
        puts("Missing Arguments Usage : progname count [display]" );
        exit(1) ;
        }

if ( argc  > 2 && !strcmp( argv[2], "display" ) )
        disp = 1 ;
else
        disp = 0 ;

for ( count = atoi( argv[1] ) ; count ; count-- )
        if ( disp )
                printf( "%d\n", count ) ;
printf( “done\n” ) ;
}
```

NB : C has a broad range of functions to convert strings into the standard data types and vice versa. For example atoi() converts a string to an integer above - remember all command line arguments are just character strings.

6.8 Dynamic Memory Allocation

This is the means by which a program can obtain and release memory at run-time. This is very important in the case of programs which use large data items e.g. databases which may need to allocate variable amounts of memory or which might have finished with a particular data block and want to release the memory used to store it for other uses.

The functions **malloc()** and **free()** form the core of C's dynamic memory allocation and are prototyped in <malloc.h>. malloc() allocates memory from the heap i.e. unused memory while available and free() releases memory back to the heap.

The following is the prototype for the malloc() function

```
void * malloc( size_t num_bytes ) ;
```

malloc() allocates num_bytes bytes of storage and returns a pointer to type void to the block of memory if successful, which can be cast to whatever type is required. If malloc() is unable to allocate the requested amount of memory it returns a NULL pointer.

For example to allocate memory for 100 characters we might do the following

```
#include <malloc.h>

void main()
{
char *p ;

if (  !( p = malloc( sizeof( char  ) * 100  ) )
        {
        puts( "Out of memory" ) ;
        exit(1) ;
        }
}
```

The return type void * is automatically cast to the type of the lvalue type but to make it more explicit we would do the following

```
if (  !( (char * )p = malloc( sizeof( char  ) * 100  ) )
        {
        puts( "Out of memory" ) ;
        exit(1) ;
        }
```

To free the block of memory allocated we do the following

```
free ( p ) ;
```

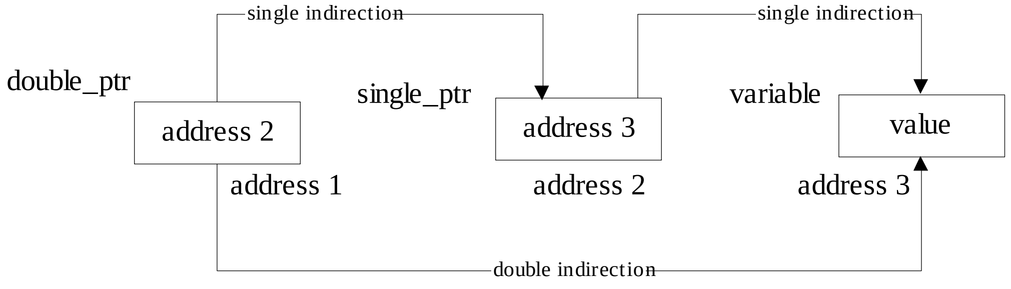
Note :- There are a number of memory allocation functions included in the standard library including calloc( ), _fmalloc( ) etc. Care must be taken to ensure that memory allocated with a particular allocation function is released with its appropriate deallocation function, e.g. memory allocated with malloc() is freed only with free() .

6.9 Multiple Indirection -- Pointers to Pointers

It is possible in C to have a pointer point to another pointer that points to a target value. This is termed multiple indirection in this case double indirection. Multiple indirection can be carried out to whatever extent is desired but can get convoluted if carried to extremes.

In the normal situation, single indirection, a pointer variable would hold the address in memory of an appropriate variable, which could then be accessed indirectly by de-referencing the pointer using the * operator.

In the case of double indirection, we have the situation where a variable may be pointed to by a pointer as with single indirection, but that this pointer may also be pointed to by another pointer. So we have the situation where we must de-reference this latter pointer twice to actually access the variable we are interested in. De-referencing the *pointer to a pointer* once gives us a normal singly indirected pointer, de-referencing the pointer to a pointer secondly allows us to access the actual data variable. The situation is depicted in the diagram below.



To declare a pointer to a pointer we include another indirection operator

```
float  * * ptr ;
```

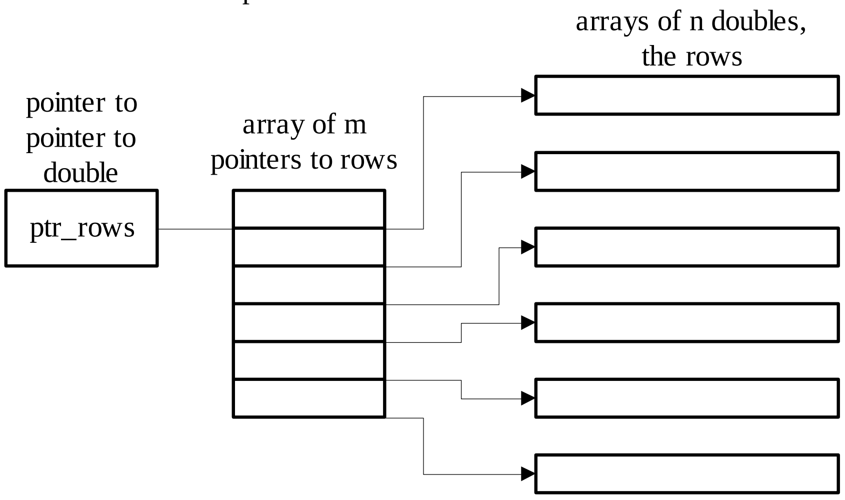
which in this case defines a *pointer to a pointer to type float*.

The following illustrates some valid operations using double indirection.

```
int x = 10, *p, **q ;

p = &x ;
q = &p ;

**q = 20 ;          // de-reference twice to access value
p = *q ;            // de-reference q once to get a pointer to int
…
int array1[] = { 1,2,3,4,5,6 ,7 ,8,9,10} ;
int array2[] = {10,20,30,40,50} ;
int *pointers[2] ;                        // an array of pointers to type int
int **ptr ;                               // a doubly indirected pointer

ptr = pointers ;              // initialise pointer to array of pointers
*ptr++ = array1 ;             // now we simply de-reference the pointer to a pointer
*ptr = array2 ;               // once and move it on like any pointer

**ptr = 100 ;                 // ptr is pointing at pointers[1] which in turn is pointing
                            // at array2 so array2[0] is assigned 100
```

For Example :- Allocation and initialisation of an m x n matrix using double indirection

What we require here is to allocate an n x n matrix as a collection of discrete rows rather than just as one block of memory. This format has advantages over a single block allocation in certain situations. The structure we end up with is illustrated below.


arrays of n doubles, the rows
pointer to pointer to double
array of m pointers to rows
ptr_rows

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

void main( void )
{
double **ptr_rows, **user_ptr, *elem_ptr ;
int m, n, i, j ;

printf( "\n\nEnter the number of rows and columns required (m, n) : " ) ;
scanf( "%d, %d", &m, &n ) ;
_flushall() ;

ptr_rows = ( double **) malloc( m * sizeof ( double * ) ) ;                    // space for row
pointers

user_ptr = ptr_rows ;
for ( i = 0; i < m ; i++ )                                                      //     and     then     row
elements
        {
        *user_ptr = (double *) malloc( n * sizeof( double ) ) ;

        elem_ptr = *user_ptr ;
        for ( j = 0; j < n ; j++ )
                *elem_ptr++ = 1.0 ;

        user_ptr++ ;                    // move onto next row pointer
        }

                                        // after use we need to clean up in reverse order
user_ptr = ptr_rows ;

for ( i = 0; i < n; i++ )
        free( *user_ptr ++ ) ;          // free a row and move onto next

free( ptr_rows ) ;                      // free pointers to rows

}
```

6.10 Pointers to Functions

A function even though not a variable still has a physical address in memory and this address may be assigned to a pointer. When a function is called it essentially causes an execution jump in the program to the location in memory where the instructions contained in the function are stored so it is possible to call a function using a pointer to a function.

The address of a function is obtained by just using the function name without any parentheses, parameters or return type in much the same way as the name of an array is the address of the array.

A pointer to a function is declared as follows

Syntax : **ret_type (* fptr) (parameter list) ;**

where fptr is declared to be a pointer to a function which takes parameters of the form indicated in the parameter list and returns a value of type ret_type.

The parentheses around * *fptr* are required because without them the declaration

```
ret_type  * fptr( parameter list ) ;
```

just declares a function fptr which returns a pointer to type ret_type !

To assign a function to a pointer we might simply do the following

```
int (*fptr)( ) ;

fptr = getchar ;  /* standard library function */
```

To call the function using a pointer we can do either of the following

```
ch = (*fptr)( ) ;
ch = fptr( ) ;
```

Example :- Program to compare two strings using a comparison function passed as a parameter.

```
#include <stdio.h>
#include <string.h>
void check( char *a, char *b, int ( * cmp ) ( ) );

void main( )
{
char s1[80], s2[80] ;
int (*p)( ) ;

p = strcmp ;

gets(s1) ;
gets( s2 );

check( s1, s2, p ) ;
}
void check ( char *a, char *b, int (* cmp)( ) )
```

```
{
if ( ! cmp( a, b ) )
        puts( "equal" ) ;
else
        puts( "not equal") ;
}
```

Note that even though we do not specify parameters to the function pointer in the prototype or declarator of the function we must specify them when actually calling the function.

Note also that instead of using an explicitly declared pointer variable to call the required function in main() we could make the call as follows

```
check( s1, s2, strcmp ) ;
```

where we essentially pass a constant pointer to strcmp( ).

For Example : Program that may check for either numeric or alphabetic equality.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check( char *a, char *b, int ( * cmp ) ( ) );
int numcmp( char *, char * ) ;

void main( )
{
char s1[80], s2[80] ;

gets(s1) ;
gets( s2 );

if ( isalpha( *s1 )                     // should have a more rigorous test here
        check( s1, s2, strcmp ) ;
else
        check( s1, s2, numcmp ) ;
}
void check ( char *a, char *b, int (* cmp)( ) )
{
if ( ! cmp( a, b ) )
        puts( "equal" ) ;
else
        puts( "not equal") ;
}

int numcmp( char *a, char *b )
{
if ( atoi( a ) == atoi( b ) )
        return 0 ;
else
        return 1 ;
}
```

6.11 Efficiency Considerations

When used correctly pointers can lead to more efficient code in situations where sequential operations on contiguous blocks of memory are required.

For example when accessing each element of an array sequentially. The inefficient way to do this is

```
for ( k = 0; k < 100; k++ )
        array[ k ] = 0.0 ;
```

When done this way the compiler has to index into the array for each iteration of the loop. This involves reading the current value of the index, k, multiplying this by the sizeof(double) and using this value as an offset from the start of the array.

The exact same thing occurs if we use a pointer incorrectly as follows

```
ptr = array ;
for ( k = 0; k < 100; k++ )
        *( ptr + k ) = 0.0 ;
```

whereas the most efficient solution is of course to do the following where the pointer itself is moved by the appropriate amount.

```
ptr = array ;
for ( k = 0; k < 100; k++ )
        *ptr++ = 0.0 ;
```

In this case we just incur the addition of sizeof(double) onto the address contained in the pointer variable for each iteration.

6.12 Exercises

1. Write your own functions to carry out some of the standard string manipulation tasks included in the standard library, e.g.. strcpy(), strcat(), strlen(), strstr( ).

Use the same prototypes as the standard functions but use slightly different function names in order to be able to test the functions in parallel with the originals.

2. There is a logical bug in the following program. Compile run and debug the program and see if
you can find it.

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

void main()
{
char  *p1 , s[80] ;

p1 = s ;
do
        {
        gets( s ) ;

        while( *           p1 )
                printf("%d",*p1++ ) ;

        } while(strcmp(s, "done") ) ;
}
```

3. The following program section contains a logical error. Run the program and explain what happens.

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

void main()
{
char *p1 = "abc", *p2 = "pacific sea" ;

printf("%s %s %s\n",p1, p2, strcat(p1, p2) );
}
```

4. Use a simple bubble sort algorithm to sort an array of characters into alphabetical order.

The bubble sort algorithm is as follows. Starting at the beginning of the array examine the first pair of adjacent elements and exchange their values if not in the correct order. Next move on by one element to the next pair of values - the pairs will overlap with the previous pair so that the second element of the first pair becomes the first element of the second pair. Examine the ordering of this pair as before. After the first pass of the bubble sort over the complete array the last array element is in the correct position, after the second pass the second last is in position, etc. Thus after each pass the unsorted portion of the array contains one less element. This procedure should be repeated until either a complete pass in which no swaps are required is made or in the worst case until one less than the total number of elements number of passes have been made.

5. Write a program which sets up an array of pointers to char which are initialised to days of the week say. Write and test a function which takes an integer parameter to print out the corresponding
day .
e.g.. 1 --- Sunday, 2 --- Monday, etc.

The function should print out an invalid message if an incorrect number is passed to it.

6. Write and test a program to sort a list of words into alphabetical order. The standard library string handling functions may be used. A maximum number of words to be sorted and a maximum length of each word should be specified in the interest of memory requirements e.g. limit the number of words to 5 and the length of each word to 10 .

7. Write and test a function which can perform unsigned integer addition with up to 50 digits of precision maintained.

For Example :-

```
      9123412341234123412341234123412341234
+     1123412341234123412341234123412341234
_________________________________________

     10246824682468246824682468246824682468
```

The program should read in the two operands and present the result of the operation as well as the two operands.

Note : There is no C data type which can maintain up to 50 digits of precision.

8. Write a simple function e.g. to add two numbers and define a pointer to this function to call the
function.

9. Write a program which will calculate and print out the values of various trigonometric operations from 0 rad to 2*PI rad at intervals of PI/4.

The trigonometric operations supported (sin, cos, tan , etc.) should be entered as command line parameters. e.g.

```
A:\>trigprog  sin
```

A single generic function should be used to compute and print out the values of the operation requested taking a pointer to the appropriate standard library function as a parameter.

Prototypes for the C standard library maths functions are included in <math.h>.

10. Write a program which will sort an array of integers using the standard library qsort() function. The qsort() function requires a pointer to a user supplied comparison function as a parameter which might have the following prototype

```
int comp( int *p1, int  *p2 ) ;
```

A return value of 0 indicates equality, a return value of 1 indicates element *p1 > element *p2, and
a return value of -1 indicates element *p1 < element *p2.

11. Programming Assignment : Base Conversion Program

Your task in this assignment is to write a C program which converts an **unsigned short** in a specified number base, base1, to another number base, base2.

The number to be converted is to be entered into the program as an **ASCII string** and the two bases, base1 and base2, as integers in base 10. The program must then convert the number in base1 to base2 and output the results for the user, continuing until an EOF is encountered (see later).

For Example :- the following input should cause an output statement like that presented below.

```
12  10  2
12 in base 10 is 1100 in base 2
```

Only base values in the range 2 -- 10 inclusive are to be allowed in the program, so you should check

1. that base1 and base2 are appropriate base values and
2. that the number string entered does not contain any invalid digits, i.e. digits outside the range for base1 (or indeed any alphabetic characters, but it may skip leading white-space characters)

before attempting any conversions. In either case your program should ignore the values completely and write out an appropriate error message to the output and then continue with the program.

To convert a number in base1 to base2 it may be simpler to carry out the operation in two steps as follows

1. Convert number string in base1 to number in base 10 and check that it is in range for unsigned short.
2. Convert base 10 number to base2 number string.

To carry out these two tasks you should provide two functions with the following prototypes

```
unsigned short convert_10( char *ptr, unsigned short base_b ) ;
char *convert_b( unsigned short num_10, unsigned short base_b );
```

The convert_10( ) function takes a pointer to an ASCII string which contains a number in base_b and should return this number as a base 10 unsigned integer.

For Example :- 23 in base 4 = $2 \times 4^1 + 3 \times 4^0 = 8 + 3 = 11$ in base 10.

Your function should check that the value does not exceed the value for an unsigned short and return an error value if it does. (Suggestion : Make 0 -- 65534 valid values and use 65535 as an error return.)

The function convert_b( ) takes a valid integer in base 10 as its first parameter and converts it to a number in base_b, which you must represent as an ASCII string.

Your function should return a pointer to a string containing the number in base_b, which should be tailored down so that the bare minimum of space is used (i.e. dynamically allocated once the number has been converted). This pointer may be freed in the calling function once it has finished with the number string in question.

For Example : Convert 12 base 10 to base 2 as follows

$\Rightarrow$ 1100 in base 2

$$2\overline{)12} = 6 \text{ Rem } 0$$
$$2\overline{)6} = 3 \text{ Rem } 0$$
$$2\overline{)3} = 1 \text{ Rem } 1$$
$$2\overline{)1} = 0 \text{ Rem } 1$$

† Your program should be written so that it accepts a number of optional command line arguments as follows

```
progname  [source_file]  [ destination_file]
```

If no command line arguments are present the program should read its input from stdin, the keyboard and write the results to stdout, the monitor.

If one command line argument is present it means that the program is to take its input from the file specified if possible where it is arranged as follows

```
12  10  2
24  5  7
```

and write the results to the standard output.

Lastly if two arguments are given the program is to read its input from the specified file and write the results to the specified destination file.

In all cases your program is to continue reading input from the input device in question until an EOF is encountered (or until a conversion error occurs). Note that EOF may be produced at the keyboard by Ctrl+Z.

Note : You may not use any of the following family of functions in your program for any reason unless you implement your own versions of them.

atoi( ), atol( ), itoa( ), ltoa( ), etc.

† File I/O is not covered until Chapter 8 so this section should be put aside until later.

12. Programming Assignment : Tabulation Program.

Your main task in this assignment is to write and test a function

int *tabulate(int *data, int numvals) ;

This function takes as its parameters a pointer to an array of integers and the number of integers in this array and should tabulate the values in the array storing the results in a second array and returning it as the functions return value.

- Your main program is to first read in a user specified number of integers from the keyboard and store them in a dynamically allocated array and then call the tabulate function.

- The tabulate function must first determine the range of the data being passed to it i.e. the maximum and minimum values it contains and then allocate a block of memory covering this range. For example if the data set was { 6, 10, 6, 8, 7 } then min = 6 and max = 10 so we need to allocate a block of (max - min + 1) = 5 integers to contain all values in the range. The first element in the array is for the value 6, the second for 7, etc.

- We also need to store the max and min range values which tells us how many data values are in the array. To do this we modify the above representation so that we allocate two extra elements to store the range. The first element in the array is deemed to be for the minimum, the second for the maximum and the remainder for the data as before.

- This new block of data is next initialised to zero i.e. no instances of any of the values in the range are present in the data. The data would look as follows with our above example.

min	max	# 6’s	# 7’s	# 8’s	# 9’s	# 10’s
6	10	0	0	0	0	0

- The function should now look at the data set and increment the count associated with all the elements in it appropriately. The data set should now look like the following and will be returned to the main program where the results will be displayed.

min	max	# 6’s	# 7’s	# 8’s	# 9’s	# 10’s
6	10	2	1	1	0	1

- Your program is also required to accept one extra command line argument. If this can be opened as a file the results of the tabulation are to be printed to it otherwise they are to be printed to stdout.

13. Programming Assignment : Polynomial Parser

The basic requirement of this assignment is to write a C program which can resolve a simple polynomial into a vector of its components and to make use of this form to implement differentiation of the equation.

The equation parser should be implemented as a function which is passed an ASCII character string which is used to represent a polynomial of degree n which has the general form

$$a_{-n}x^{-n} + a_{-n+1}x^{-n+1} + \cdots a_{-1}x^{-1} + a_0 + a_1x + a_2x^2 + \cdots a_{n-1}x^{n-1} + a_nx^n$$

where x is the dependant variable, a_n are the coefficients of the polynomial (which may be limited to signed integer values).

The character string representation may be of any degree, n, and the components may be in any random order and may even be repeated. Use the '^' character to represent *to the power of.* The character string may also include white space characters for readability, namely the space character, which do not convey any information but which nonetheless must be processed being part of the string. Some examples of valid equations are given below where '_' represents the space character.

```
1 + x + 2x^2 + 5x^4
2x^4 + 3x -12
-6 + 2x^3 - 2x + 4 - x
-4 x^-2 + x^-1+ 2x^3
x_^_2_+_2_x_^_3
```

The equation parser must resolve the polynomial into a vector representation, ie. it must establish a value for the coefficient of each component of the polynomial. This information or vector should be stored in a single dimension integer array where the index into the array represents a specific component of the vector. Vector[0] represents the coefficient of x^0, Vector[1] represents the coefficient of x^1, and so on up to the degree of the polynomial.

The size of the vector array should be just sufficient to store the required vector, memory being allocated dynamically once the degree of the polynomial has been established. For example a polynomial of degree 2 would require an array of three elements to store all the component coefficients.

For testing purposes your program should first of all take in a polynomial in the form of a character string as input, resolve this into a vector using the parser function, and then pass this vector to a function which should differentiate the vector (or equation) and store the result in a different vector of appropriate dimension.

The results of each operation should be displayed in tabular form as illustrated in the following example where the equation input was 5x^3 + 3x^2 - x + 4.

Coefficients :	x^0	x^1	x^2	x^3
Equation :	4	-1	3	5
Equation' :	-1	6	15	

The results of the operation should also be stored to a user specified text file in the same tabular form as above. In the case where the file specified already exists and has a number of valid entries, the new entry to be added should be separated by a blank line from the others for readability.

Chapter 7

Structures & Unions

7.1 Structures

A structure is a customised user-defined data type in C. It is by definition a collection of variables of any type that are referenced under one name, providing a convenient means of keeping related information together.

Some terminology :-

structure definition :- the template used to create structure variables.
structure elements :- the member variables of the structure type

Defining Structures

Syntax :

```
struct  tag   {
        type  var_1 ;
        type var_2 ;
        ...
        type var_n ;
        }  ;
```

The keyword **struct** tells the compiler we are dealing with a structure and must be present whenever we refer to the new type, **tag** is an identifier which is the name given to the customised "type".

A variable of this new type can now be defined as follows for example. Note that the keyword struct has to be used in conjunction with our own name for the structure, *tag*.

```
struct tag variable ;
```

For Example :-

```
struct RECORD {
        int rec_no ;
        char name[30] ;
        char town[40] ;
        char country[ 20 ]
        } ;
struct RECORD person ;
```

The compiler will automatically allocate enough storage to accommodate all the elements. To find out how much storage is required one might do the following

```
size = sizeof ( person ) ;
```

or

```
size = sizeof( struct RECORD ) ;
```

NB : The name of a structure is not the address of the structure as with array names.

Accessing Structure Elements

For example define a complex type structure as follows.

```
struct complex {
        double real ;
        double imaginary ;              // Note that a variable may also be
        } cplx ;                        // defined at structure definition time
```

The elements of the structure are accessed using the **dot operator**, **.** , as follows

```
cplx.real  =  10.0 ;
cplx.imag  =  20.23 ;
scanf ( "%lf", &cplx.real ) ;
```

or if we want to access struct RECORD already defined

```
puts( person.name ) ;
```

or character by character

```
person.name[i] = 'a' ;
```

Thus we treat structure elements exactly as normal variables and view the dot operator as just another appendage like the indirection operator or an array index.

Initialising Structures

Structure elements or fields can be initialised to specific values as follows :-

```
struct id {
        char name[30] ;
        int id_no ;
        } ;
struct id student = { "John", 4563 } ;
```

Structure Assignment

The name of a structure variable can be used on its own to reference the complete structure. So instead of having to assign all structure element values separately, a single assignment statement may be used to assign the values of one structure to another structure of the same type.

For Example :-

```
struct {
        int a, b ;
        }  x = {1, 2 }, y ;

y = x ;          // assigns values of all fields in x to fields in y
```

Creating more Complex Structures with Structures

Once again emphasising that structures are just like any other type in C we can create arrays of structures, nest structures, pass structures as arguments to functions, etc.

For example we can nest structures as follows creating a structure employee_log that has another structure as one of its members.

```
struct time {
        int hour ;
        int min ;
        int sec ;
        } ;
struct employee_log {
        char name[30] ;
        struct time start, finish ;
        } employee_1 ;
```

To access the hour field of time in the variable employee_1 just apply the dot operator twice

```
employee_1.start.hour = 9 ;
```

Typically a company will need to keep track of more than one employee so that an array of *employee_log* would be useful.

```
struct employee_log  workers[100] ;
```

To access specific employees we simply index using square braces as normal, e.g. workers[10]. To access specific members of this structure we simply apply the dot operator on top of the index.

```
workers[10].finish.hour = 10 ;
```

When structures or arrays of structures are not global they must be passed to functions as parameters subject to the usual rules. For example

```
function1( employee_1 ) ;
```

implements a call to function1 which might be prototyped as follows

```
void  function1( struct employee_log emp ) ;
```

Note however that a full local copy of the structure passed is made so if a large structure is involved memory the overhead to simply copy the parameter will be high so we should employ call by reference instead as we will see in the next section.

Passing an array of structures to a function also follows the normal rules but note that in this case as it is impossible to pass an array by value no heavy initialisation penalty is paid - we essentially have call by reference. For example

```
function2( workers ) ;
```

passes an array of structures to *function2* where the function is prototyped as follows.

```
function2( struct employee_log staff[ ] ) ;
```

Structure Pointers

As we have said already we need call by reference calls which are much more efficient than normal call by value calls when passing structures as parameters. This applies even if we do not intend the function to change the structure argument.

A structure pointer is declared in the same way as any pointer for example

```
struct address {
        char name[20] ;
        char street[20] ;
        } ;
struct address person ;
struct address *addr_ptr ;
```

declares a pointer addr_ptr to data type *struct address*.

To point to the variable person declared above we simply write

```
addr_ptr =  &person ;
```

which assigns the address of person to addr_ptr.

To access the elements using a pointer we need a new operator called the arrow operator, ->, which can be used **only** with structure pointers. For example

```
puts( addr_ptr -> name ) ;
```

For Example :- Program using a structure to store time values.

```
#include <stdio.h>

struct time_var {
        int hours, minutes, seconds ;
        } ;
void display ( const struct time_var * ) ;          /* note structure pointer and const */

void main()
{
struct time_var time ;

time.hours = 12 ;
time.minutes = 0 ;
time.seconds = 0 ;
display( &time ) ;
}
void display( const struct time_var *t )
{
printf( "%2d:%2d;%2d\n", t -> hours, t -> minutes, t -> seconds ) ;
}
```

Note that even though we are not changing any values in the structure variable we still employ call by reference for speed and efficiency. To clarify this situation the *const* keyword has been employed.

Dynamic allocation of structures

The memory allocation functions may also be used to allocate memory for user defined types such as structures. All malloc() basically needs to know is how much memory to reserve.

For Example :-

```
                    struct coordinate {
                              int x, y, z ;
                              } ;
                    struct coordinate *ptr ;

                    ptr = (struct coordinate * ) malloc( sizeof ( struct coordinate )  ) ;
```

7.2 Bit--Fields

Bit--fields are based directly on structures with the additional feature of allowing the programmer to specify the size of each of the elements in bits to keep storage requirements at a minimum. However bit--field elements are restricted to be of type int (signed or unsigned).

For Example :-

```
                    struct clock {
                              unsigned hour : 5 ;
                              unsigned minutes : 6 ;
                              unsigned seconds : 6 ;
                              } time ;
```

This time structure requires 17 bits to store the information now so the storage requirement is rounded up to 3 bytes. Using the normal structure format and 32-bit integer elements we would require 12 bytes so we achieve a substantial saving.

Bit--fields can be used instead of the bitwise operators in system level programming, for example to analyse the individual bits of values read from a hardware port we might define the following bit-field.

```
                    struct status {
                              unsigned bit0 : 1 ;
                              unsigned bit1 : 1 ;
                              ...
                              unsigned bit15 : 1 ;
                              } ;
```

If we are interested in bit 15 only we need only do the following

```
                    struct status {
                              unsigned : 15 ;            /* skip over first 15 bits with a "non-member"
*/
                              unsigned bit15 : 1 ;
                              } ;
```

There are two further restrictions on the use of bit--fields

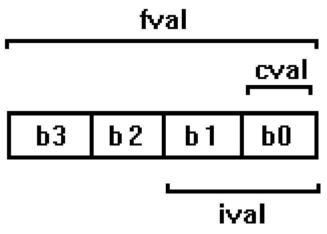
- Cannot take the address of a bit--field variable
- Cannot create an array of bit--field variables

Can you suggest reasons for this ?

7.3 Unions

A union is data type where the data area is shared by two or more members generally of different type at different times.

For Example :-



```
union u_tag {
    short ival ;
    float fval ;
    char cval ;
    } uval ;
```

The size of uval will be the size required to store the largest single member, 4 bytes in this case to accommodate the floating point member.

Union members are accessed in the same way as structure members and union pointers are valid.

```
uval.ival = 10 ;
uval.cval = 'c' ;
```

When the union is accessed as a character we are only using the bottom byte of storage, when it is accessed as a short integer the bottom two bytes etc. It is up to the programmer to ensure that the element accessed contains a meaningful value.

A union might be used in conjunction with the bit-field *struct status* in the previous section to implement binary conversions in C.

For Example :-

```
union conversion {
        unsigned short num ;
        struct status bits ;
} number ;
```

We can load number with an integer

```
scanf( "%u", &number.num );
```

Since the integer and bit--field elements of the union share the same storage if we now access the union as the bit--field variable *bits* we can interpret the binary representation of num directly.

i.e.

```
if ( uvar.bits.bit15 )
        putchar( '1' ) ;
else
        putchar('0') ;
...
if ( uvar.bits.bit0 )
        putchar( '1' ) ;
else
        putchar('0') ;
```

Admittedly rather inefficient and inelegant but effective.

7.4 Enumerations

An enumeration is a user defined data type whose values consist of a set of named integer constants, and are used for the sole purpose of making program code more readable.

Syntax: **enum tag { value_list } [enum_var] ;**

where tag is the name of the enumeration type, value_list is a list of valid values for the enumeration, and where enum_var is an actual variable of this type.

For Example :-

```
enum colours { red, green, blue, orange } shade ;
                        //  values red - 0, green - 1, blue - 2, orange - 3

enum day { sun = 1, mon, tue, wed = 21, thur, fri, sat } ;
enum day weekday ;
                        //  values are 1, 2, 3, 21, 22, 23, 24
```

Variables declared as enumerated types are treated exactly as normal variables in use and are converted to integers in any expressions in which they are used.

For Example :-

```
int i ;

shade = red ;     // assign a value to shade enum variable

i = shade ;       // assign value of enum to an int

shade = 3 ;       // assign valid int to an enum, treat with care
```

7.5 The typedef Keyword

C makes use of the **typedef** keyword to allow new data type **names** to be defined. No new type is created, an existing type will now simply be recognised by another name as well. The existing type can be one of the in-built types or a user-defined type.

Syntax : **typedef type name ;**

where type is any C data type and name is the new name for this type.

For Example :-

```
typedef  int INTEGER ;
INTEGER i ;                   // can now declare a variable of type ‘INTEGER’

typedef double * double_ptr ;
double_ptr  ptr ;              //  no need of * here as it is part of the type

typedef struct coords {
        int x, y ;
        } xycoord ;            //  xycoord is now a type name in C
xycoord  coord_var ;
```

The use of typedef makes program code easier to read and when used intelligently can facilitate the porting of code to a different platform and the modification of code. For example in a first attempt at a particular program we might decide that floating point variables will fill our needs. At a later date we decide that all floating point variables really should be of type double so we have to change them all. This problem is trivial if we had used a typedef as follows :-

```
typedef float FLOATING ;
```

To remedy the situation we modify the user defined type as follows

```
typedef double FLOATING ;
```

7.6 Linked Lists

When we have needed to represent collections of data up until now we have typically used data structures such as arrays whose dimensions were fixed at compile time or which were dynamically allocated as required. These arrays could have been arrays of basic data types, e.g. doubles, or could have been arrays of structures when more complex data needed to be represented.

However this method of representing data, while perfectly adequate in many situations, is limited and can be very inefficient when we are dealing with situations where the data collection has to be modified at run-time.

For example if we are dealing with an ordered list of records and we need to insert a new record, for John say, in the correct position in the list we might have to go through the following steps.

1. Expand the memory space allocated to the array to take one more record.

Alan	Ben	Sam	Tom	

2. Determine the correct position in the array at which to insert the element – position 3.

3. Reposition the data so that the appropriate position is left unused.

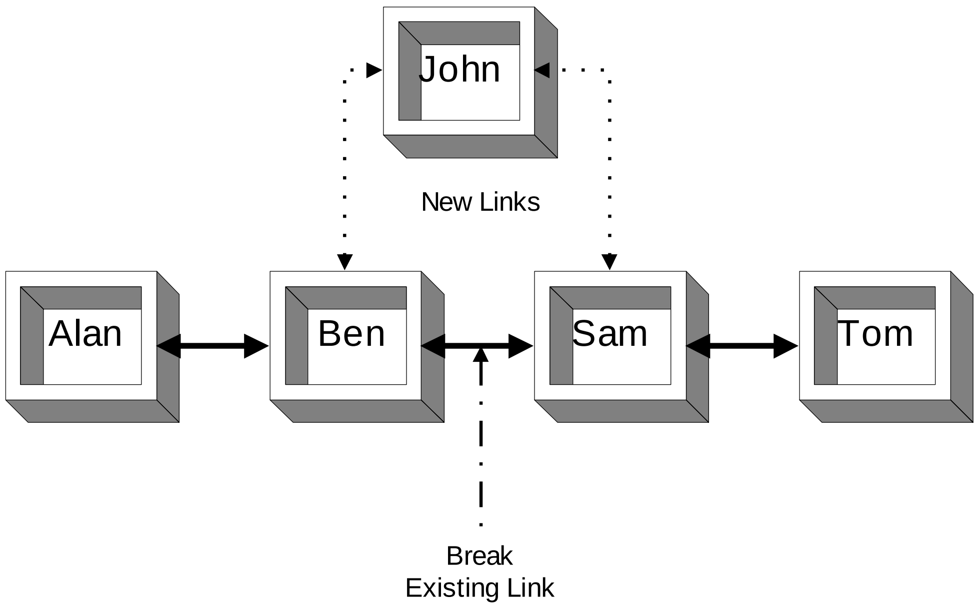
Alan	Ben		Sam	Tom

4. Insert the new element into the list.

Alan	Ben	John	Sam	Tom

Whatever our internal representation of the data, steps 1,2 & 4 will have to be carried out in one form or another. However step 3 can be avoided if we represent the data in an alternate fashion.

If we can represent our data as a chain of inter-connected records as illustrated below where each record has its own memory space and knows which record comes before and after it we have a much more flexible structure.



Now when we want to insert a new record in position we still must allocate the appropriate amount of memory to store it (note in this case it is a discrete block) and we still must figure out which position it belongs in but we no longer have to manipulate the position of the other records in memory. These will now stay exactly where they are but we will have to inform the records before and after the position at which we wish to insert that they will be pointing to a different record.

Using the normal terminology we say we have a list of nodes linked together by pointer links. To insert a new item into the list we break an existing link and create two new links incorporating the new node into the list.

Nodes or Self-Referential Structures

A structure that contains a pointer member that points to a structure of the same type as itself is said to be a self-referential structure. For example :-

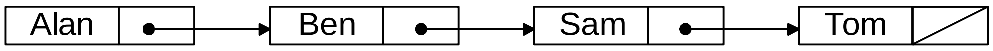
```
struct node {
        char name[20] ;
        struct node *nextnode ;
};
```

This defines a new type, **struct node**, which has a pointer member, *nextnode*, which points to a structure of the same type as itself. This node pointer allows this current node to be linked to another and so a list of linked nodes can be built up.

Note that even though the structure is not fully defined when the *nextnode* field is specified the compiler does not have a problem as all it needs to know is that there is such a type.

The above definition of struct node allows us to build a singly linked list i.e. each node only knows where the node following it is located. A null pointer is used to indicate the end of a linked list structure.

The diagram below illustrates how this node structure would be used to represent our linked list above. Each node contains two fields the actual data and a pointer link to the next node. The final node in the list contains a null pointer link as indicated by the slash in the diagram.



Connecting the Nodes

The basic building block in a linked list is just a C structure so we can initialise it in the same way as any other structure. For example in our above situation to allocate and initialise the first node we might do the following :-

```
struct node *node_1 ;

node_1 = ( struct node *)malloc( sizeof ( struct node ) ) ;

strcpy( node_1->name, “Alan” ) ;
node_1->nextnode = NULL ;
```

This block of code initialises this node only and because the *nextnode* pointer is assigned a null pointer value it designates it as the final (and only) node in the list.

If we want to add on an item to the list we can do the following :-

```
struct node *node_2 ;

node_2 = ( struct node *)malloc( sizeof ( struct node ) ) ;

strcpy( node_2->name, “Ben” ) ;
node_2->nextnode = NULL ;
node_1->nextnode = node_2 ;
```

Note that it is only the last line of code that adds the item onto the list. The remaining code just allocates storage and initialises a new node as before.

We now have a linked list with two nodes. The node pointer *node_1* points to the first element in the list which is normally called the list **head**. As all the nodes in the list are connected via their link pointer this is in fact a pointer to the complete linked list so we wouldn’t need to store node_2, etc. at all. The last node has a null pointer which identifies it as the last node in the linked list and is called the list **tail**.

Operations on Linked Lists

When working with linked lists we will need to perform a number of operations quite often so it makes sense to build up a library of these at the outset. We will need to be able to insert items into the list, remove items from the list, traverse the list, etc.

We will continue to use our definition of a node except that we will typedef it as follows :-

```
typedef struct node {
        char name[20] ;
        struct node *nextnode ;
} list_node ;
```

Note in this case we cannot use the *typedef* name to declare nextnode as this name is not in existence at that point.

To declare the linked list we simply declare a pointer to one of these structure types.

```
list_node *ptr_head = NULL ;
```

Traversing a List

In many list operations we will need to process the information in each node, for example to print the complete information in the list; this is termed *traversing a list.*

An iterative version of a *print_list* function might be as follows :-

```
void print_list( list_node *p_head )
{
list_node *node_ptr ;

for ( node_ptr = p_head ; node_ptr != NULL ; node_ptr = node_ptr->nextnode )
        puts( node_ptr->name ) ;
}
```

whereas a recursive version might be implemented as follows :-

```
void print_list( list_node *p_head )
{
if ( p_head == NULL )
        puts("\n" );
else
        {
        puts( p_head->name ) ;
        print_list( p_head->nextnode ) ;
        }
}
```

Inserting a Node

There are a variety of ways this function can be implemented - inserting a node in the correct order, at the head, at the tail, by position, etc. For our example we will write a function to insert a new node in the correct order assuming the node has been fully created before being passed to us.

```
void insert( list_node **p_head, list_node *new )
{
list_node *node_ptr = *p_head ;
list_node *prev = NULL ;

while (  node_ptr != NULL  && (strcmp( new->name, node_ptr->name ) > 0 ) )
        {
        prev = node_ptr ;
        node_ptr = node_ptr->nextnode ;
        }

if ( prev == NULL )                // head position
        {
        new->nextnode = *p_head ;
        *p_head = new ;
        }
else
        {
        prev->nextnode = new ;
        new->nextnode = node_ptr ;
        }
    }
```

Removing a Node

We will implement this as a function which finds the node by the name field, unlinks it from the list and returns a pointer to the removed node without freeing its associated memory.

```
list_node *delete( list_node **p_head, char *name )
{
list_node *prev = NULL, *curr, *node_ptr = *p_head ;

while (  node_ptr != NULL  && strcmp( name, node_ptr->name ) )
        {
        prev = node_ptr ;
        node_ptr = node_ptr->nextnode ;
        }
if ( prev == NULL )                  // removing head position
        {
        *p_head = (*p_head)->nextnode ;
        return node_ptr ;
        }
else    {
        prev->nextnode = node_ptr->nextnode ;
        return node_ptr ;
        }
}
```

Note the use of double indirection to type *list_node* in each of the above functions. Can you explain why this is necessary ?

The following main function is a trivial illustration of how to use the above functions in practice.

```
void main()
{
list_node *ptr_head = NULL ;
struct node *node_1, *node_2 ;

node_1 = ( struct node *)malloc( sizeof ( struct node ) ) ;
strcpy( node_1->name, "Alan" ) ;
insert( &ptr_head, node_1 ) ;

node_2 = ( struct node *)malloc( sizeof ( struct node ) ) ;
strcpy( node_2->name, "Ben" ) ;
insert( &ptr_head, node_2) ;

puts( "List with Alan & Ben") ;
print_list( ptr_head ) ;

delete( &ptr_head, "Ben" ) ;

puts( "List with Alan") ;
print_list( ptr_head ) ;
}
```

What we have discussed in the example above is one of the more general variations of a linked list. Many different implementations can be implemented with slightly different functionality - for example we can implement doubly linked lists, stacks (LIFO) or queues (FIFO) using the same basic building blocks.

7.7 Efficiency Considerations

As described previously pointers should be used to implement call-by-reference in passing large user defined data items to functions to reduce the copying overhead involved in call-by-value. If implementing a typical call-by-value situation use of the const keyword can protect us from inadvertent modification of the referenced argument.

Care should be taken not to go overboard in defining large complicated data structures without need and to use them appropriately.

For example when dealing with lists of data of any type a normal array may sometimes be a better choice as a data structure rather than a linked list. The big advantage a linked list has over a linear array is that you can insert or delete items at will without shifting the remaining data.

However the extra overhead involved in setting up the linked list, both the calls to malloc() to allocate each new element and the ‘extra’ pointer associated with each element, may not justify its selection.

Suppose we need to keep a list of integers. Then most of the storage is taken up with the extra pointer information used solely for maintaining the list (the pointer to the next element and the pointer to each integer).

7.8 Exercises

1. Write a program which will simulate the action of a digital clock continuously as closely as possible. Use a structure to hold the values of hours, minutes and seconds required. Your program should include a function to display the current time and to update the time appropriately using a delay function/loop to simulate real time reasonably well. Pointers to the clock structure should be passed to the display and update functions rather than using global variables.

2. Complex numbers are not supported as a distinct type in C but are usually implemented by individual users as user defined structures with a set of functions representing the operations possible on them. Define a data type for complex values and provide functions to support addition, subtraction, multiplication and division of these numbers. Use call by reference to make these functions as efficient as possible and try to make their use as intuitive as you can.

3. Write a program which will

(a) set up an array of structures which will hold the names, dates of birth, and addresses of a number of employees.
(b) assign values to the structure from the keyboard.
(c) sort the structures into alphabetical order by name using a simple bubble sort algorithm.
(d) print out the contents of a specific array element on demand.

4. Use a combination of bit--fields and unions in C to print out the bit patterns of any unsigned integer values input at the keyboard.

5. BCD (Binary Coded Decimal) codes are commonly used in industrial applications to represent
decimal numbers. Here four bits are used to represent each decimal digit , e.g.

```
0000  ==>   0
0001  ==>   1
...
1001  ==>   9
```

Thus short int can be used to represent a four digit decimal number.

Use Bit--Fields to implement conversion routines to encode a decimal number into BCD format and back again.

6. Redo problem 1 using a bit--field structure as an illustration of how memory requirements may
be reduced. Compare the old and new memory requirements for data storage.

7. Using a doubly linked list data structure write a program that maintains an alphabetically ordered list of student registration records. The information held in each record should include the student's name, student identity number, address, course, etc. Your program should allow the user to add on records to the list, to remove records from the list, to amend records and to produce a screen dump of all records. Use distinct functions for all major operations if possible.

8. Use a FIFO implementation of a linked list to represent a queue of people entered by the user.

Chapter 8

Standard File I/O

The C standard library I/O functions allow you to read and write data to both files and devices. There are no predefined file structures in C, all data being treated as a sequence of bytes. These I/O functions may be broken into two different categories : stream I/O and low-level I/O.

The stream I/O functions treat a data file as a stream of individual characters. The appropriate stream function can provide buffered, formatted or unformatted input and output of data, ranging from single characters to complicated structures. Buffering streamlines the I/O process by providing temporary storage for data which takes away the burden from the system of writing each item of data directly and instead allows the buffer to fill before causing the data to be written.

The low-level I/O system on the other hand does not perform any buffering or formatting of data --instead it makes direct use of the system's I/O capabilities to transfer usually large blocks of information.

8.1 Stream I/O

The C I/O system provides a consistent interface to the programmer independent of the actual device being accessed. This interface is termed a **stream** in C and the actual device is termed a **file**. A device may be a disk or tape drive, the screen, printer port, etc. but this does not bother the programmer because the stream interface is designed to be largely device independent. All I/O through the keyboard and screen that we have seen so far is in fact done through special standard streams called **stdin** and **stdout** for input and output respectively. So in essence the console functions that we have used so far such as printf(), etc. are special case versions of the file functions we will now discuss.

There are two types of streams : text and binary. These streams are basically the same in that all types of data can be transferred through them however there is one important difference between them as we will see.

Text Streams

A text stream is simply a sequence of characters. However the characters in the stream are open to translation or interpretation by the host environment. For example the newline character, '\n', will normally be converted into a carriage return/linefeed pair and ^Z will be interpreted as EOF. Thus the number of characters sent may not equal the number of characters received.

Binary Streams

A binary stream is a sequence of data comprised of bytes that will not be interfered with so that a one-to-one relationship is maintained between data sent and data received.

Common File Functions

`fopen()`	open a stream
`fclose()`	close a stream
`putc()& fputc()`	write a character to a stream
`getc()& fgetc()`	read a character from a stream
`fprintf()& fscanf`	formatted I/O
`fgets() & fputs()`	string handling
`fseek()`	position the file pointer at a particular byte
`feof()`	tests if EOF

Opening and Closing Files

A stream is associated with a specific file by performing an open operation. Once a file is opened information may be exchanged between it and your program. Each file that is opened has a unique file control structure of type FILE (which is defined in <stdio.h> along with the prototypes for all I/O functions and constants such as EOF (-1)). A **file pointer** is a pointer to this FILE structure which identifies a specific file and defines various things about the file including its name, read/write status, and current position. A file pointer variable is defined as follows

```
FILE  *fptr ;
```

The fopen() function opens a stream for use and links a file with that stream returning a valid file pointer which is positioned correctly within the file if all is correct. fopen() has the following prototype

```
FILE *fopen( const char *filename, const char *mode );
```

where filename is a pointer to a string of characters which make up the name and path of the required file, and mode is a pointer to a string which specifies how the file is to be opened. The following table lists some values for mode.

r	opens a text file for reading (must exist)
w	opens a text file for writing (overwritten or created)
a	append to a text file
rb	opens a binary file for reading
wb	opens a binary file for writing
ab	appends to a binary file
r+	opens a text file for read/write (must exist)
w+	opens a text file for read/write
a+	append a text file for read/write
rb+	opens a binary file for read/write
wb+	opens a binary file for read/write
ab+	append a binary file for read/write

If `fopen( )` cannot open "`test.dat` " it will a return a NULL pointer which should always be tested for as follows.

```
FILE *fp ;
if (  ( fp = fopen( "test.dat", "r" ) )  ==  NULL )
     {
     puts( "Cannot open file") ;
     exit( 1) ;
     }
```

This will cause the program to be exited immediately if the file cannot be opened.
The `fclose()` function is used to disassociate a file from a stream and free the stream for use again.

```
            fclose( fp ) ;
```

`fclose()` will automatically flush any data remaining in the data buffers to the file.

Reading & Writing Characters

Once a file pointer has been linked to a file we can write characters to it using the `fputc()` function.

```
            fputc(  ch,  fp ) ;
```

If successful the function returns the character written otherwise EOF. Characters may be read from a file using the fgetc() standard library function.

```
            ch =  fgetc( fp ) ;
```

When EOF is reached in the file `fgetc( )` returns the EOF character which informs us to stop reading as there is nothing more left in the file.

For Example :- Program to copy a file byte by byte

```
      #include  <stdio.h>
      void main()
      {
      FILE *fin, *fout ;
      char dest[30], source[30], ch ;

      puts( "Enter source file name" );
      gets( source );
      puts( "Enter destination file name" );
      gets( dest ) ;

      if ( ( fin = fopen( source, "rb" ) )  == NULL )        //
open as binary as we don’t
            {// know what is in file
            puts( "Cannot open input file ") ;
            puts( source ) ;
            exit( 1 ) ;
            }

      if ( ( fout = fopen( dest, "wb" ) )  == NULL )
            {
            puts( "Cannot open output file ") ;
            puts( dest ) ;
            exit( 1 ) ;
            }

      while ( ( ch = fgetc( fin ) )  !=  EOF  )
            fputc( ch , fout ) ;

      fclose( fin ) ;
      fclose( fout ) ;
      }
```

NB : When any stream I/O function such as fgetc() is called the current position of the file pointer is automatically moved on by the appropriate amount, 1 character/ byte in the case of fgetc() ;

Working with strings of text

This is quite similar to working with characters except that we use the functions `fgets()` and `fputs()` whose prototypes are as follows :-

```
      int fputs( const char *str, FILE *fp ) ;
      char *fgets( char *str, int maxlen, FILE *fp ) ;
```

For Example : Program to read lines of text from the keyboard, write them to a file and then read them back again.

```
      #include <stdio.h>
      void main()
      {
      char file[80], string[80] ;
      FILE *fp ;

      printf( "Enter file Name : " );
      gets( file );

      if (( fp = fopen( file, "w" ))  == NULL )//open for writing
            {
            printf( "Cannot open file %s", file ) ;
            exit( 1 ) ;
            }

      while ( strlen ( gets( str ) ) > 0 )
            {
            fputs( str, fp ) ;
            fputc( '\n', fp ) ;  /* must append \n for readability
-- not stored by gets() */
            }

      fclose( fp ) ;

      if (( fp = fopen( file, "r" ))  == NULL )//open for reading
            {
            printf( "Cannot open file %s", file ) ;
            exit( 1 ) ;
            }

      while (fgets( str, 79, fptr )  != EOF )// read at most 79
characters
            puts( str ) ;

      fclose( fp ) ;
      }
```

Formatted I/O

For Example :- Program to read in a string and an integer from the keyboard, write them to a disk file and then read and display the file contents on screen.

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

void main()
{
FILE *fp ;
char s[80] ;
int t ;

if ( ( fp = fopen( "test.dat", "w" ) ) == NULL )
      {
      puts( "Cannot open file test.dat") ;
      exit(1) ;
      }

puts( "Enter a string and a number") ;
scanf( "%s %d", s, &t );
fprintf( fp, "%s %d", s, t );
fclose( fp ) ;

if ( ( fp = fopen( "test.dat", "r" ) ) == NULL )
      {
      puts) "Cannot open file") ;
      exit(1) ;
      }

fscanf( fp, "%s %d" , s , &t ) ;
printf( "%s, %d\n", s, t ) ;

fclose( fp ) ;
}
```

Note : There are several I/O streams opened automatically at the start of every C program.

`stdin`	---	standard input ie. keyboard
`stdout`	---	standard output ie. screen
`stderr`	---	again the screen for use if stdout malfunctions

It is through these streams that the console functions we normally use operate. For example in reality a normal printf call such as

```
printf( "%s %d", s, t ) ;
```

is in fact interpreted as

```
fprintf( stdout, "%s %d", s, t ) ;
```

fread() and fwrite()

These two functions are used to read and write blocks of data of any type. Their prototypes are as follows where size_t is equivalent to unsigned.

```
size_t  fread( void *buffer,  size_t num_bytes,  size_t count,
FILE *fp ) ;
size_t  fwrite( const void *buffer,  size_t num_bytes,  size_t
count,  FILE *fp ) ;
```

where **buffer** is a pointer to the region in memory from which the data is to be read or written respectively, **num_bytes** is the number of bytes in each item to be read or written, and **count** is the total number of items (each num_bytes long) to be read/written. The functions return the number of items successfully read or written.

For Example :-

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

struct tag {
      float balance ;
      char name[ 80 ] ;
      } customer  = { 123.232, "John" } ;

void main()
{
FILE *fp ;
double d = 12.34 ;
int i[4] = {10 , 20, 30, 40 } ;

if ( (fp = fopen ( "test.dat", "wb+" ) ) == NULL )
      {
      puts( "Cannot open File" ) ;
      exit(1) ;
      }

fwrite( &d, sizeof( double ), 1, fp ) ;
fwrite( i, sizeof( int ), 4, fp ) ;
fwrite( &customer, sizeof( struct tag ), 1, fp ) ;

rewind( fp ) ;   /* repositions file pointer to start */

fread( &d, sizeof( double ), 1, fp ) ;
fread( i, sizeof( int ), 4, fp ) ;
fread( &customer, sizeof( struct tag ), 1, fp ) ;

fclose( fp ) ;
}
```

NB : Unlike all the other functions we have encountered so far fread and fwrite read and write **binary** data in the same format as it is stored in memory so if we try to edit one these files it will appear completely garbled. Functions like fprintf, fgets, etc. read and write displayable data. fprintf will write a double as a series of digits while fwrite will transfer the contents of the 8 bytes of memory where the double is stored directly.

Random Access I/O

The `fseek()` function is used in C to perform random access I/O and has the following prototype.

```
      int fseek ( FILE *fp, long num_bytes, int origin ) ;
```

where **origin** specifies one of the following positions as the origin in the operation

SEEK_SET	---	beginning of file
SEEK_CUR	---	current position
SEEK_END	---	EOF

and where **num_bytes** is the offset in bytes to the required position in the file. fseek() returns zero when successful, otherwise a non-zero value.

For Example if we had opened a file which stored an array of integers and we wish to read the 50th value we might do the following

```
      fseek ( fp, ( 49 * sizeof( int ) ), SEEK_SET ) ;
      fscanf( fp, "%d", &i ) ;
```

from anywhere in the program.

8.2 Low -- Level I/O

Low level file input and output in C does not perform any formatting or buffering of data whatsoever, transferring blocks of anonymous data instead by making use of the underlying operating system's capabilities.

Low level I/O makes use of a file handle or descriptor, which is just a non-negative integer, to uniquely identify a file instead of using a pointer to the FILE structure as in the case of stream I/O.

As in the case of stream I/O a number of standard files are opened automatically :-

standard input	**---**	**0**
standard output	**---**	**1**
standard error	**---**	**2**

The following table lists some of the more common low level I/O functions, whose prototypes are given in <io.h> and some associated constants are contained in <fcntl.h> and <sys\stat.h>.

open()	opens a disk file
close()	closes a disk file
read()	reads a buffer of data from disk
write()	writes a buffer of data to disk

The open function has the following prototype and returns -1 if the open operation fails.

```
      int open ( char *filename, int oflag [, int pmode] ) ;
```

where filename is the name of the file to be opened, oflag specifies the type of operations that are to be allowed on the file, and pmode specifies how a file is to be created if it does not exist.

oflag may be any logical combination of the following constants which are just *bit flags* combined using the bitwise OR operator.

O_APPEND	appends to end of file
O_BINARY	binary mode
O_CREAT	creates a new file if it doesn't exist
O_RDONLY	read only access
O_RDWR	read write access
O_TEXT	text mode
O_TRUNC	truncates file to zero length
O_WRONLY	write only access

pmode is only used when O_CREAT is specified as part of oflag and may be one of the following values

S_IWRITE
S_IREAD
S_IREAD | S_IWRITE

This will actually set the read / write access permission of the file at the operating system level permanently unlike oflag which specifies read / write access just while your program uses the file.

The `close()` function has the following prototype

```
      int close ( int handle ) ;
```

and closes the file associated with the specific handle.

The `read()` and `write()` functions have the following prototypes

```
      int read( int handle, void *buffer, unsigned int count ) ;
      int write( int handle, void *buffer, unsigned int count ) ;
```

where handle refers to a specific file opened with open(), buffer is the storage location for the data (of any type) and count is the maximum number of bytes to be read in the case of read() or the maximum number of bytes written in the case of write(). The function returns the number of bytes actually read or written or -1 if an error occurred during the operation.

Example : Program to read the first 1000 characters from a file and copy them to another.

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

void main()
{
char buff[1000] ;
int handle ;

handle=open(" test.dat", O_BINARY|O_RDONLY, S_IREAD | S_IWRITE );
if ( handle == -1 ) return ;
if ( read( handle, buff, 1000 ) == 1000 )
  puts( "Read successful");
else
```

```
  {
  puts( Read failed"  ) ;
  exit( 1 );
}

close( handle ) ;

handle = open("test.bak",
               O_BINARY|O_CREAT|O_WRONLY| O_TRUNC,
               S_IREAD | S_IWRITE  ) ;

if ( write( handle, buff, 1000 )  == 1000 )
     puts( "Write successful") ;
else
     {
     puts( "Write Failed") ;
     exit( 1 ) ;
     }

close( handle ) ;
}
```

Low level file I/O also provides a seek function **lseek** with the following prototype.

```
 long _lseek( int handle, long offset, int origin );
```

_lseek uses the same origin etc. as *fseek()* but unlike fseek() returns the offset, in bytes, of the new file position from the beginning of the file or -1 if an error occurs.

For Example : Program to determine the size in bytes of a file.

```
     #include <stdio.h>
     #include <io.h>
     #include <fcntl.h>
     #include <sys\stat.h>

     void main()
     {
     int handle ;
     long length ;
     char name[80] ;

     printf( “Enter file name : ” ) ;
     gets( name ) ;
     handle=open( name,O_BINARY| O_RDONLY, S_IREAD | S_IWRITE );

     lseek(  handle, 0L, SEEK_SET ) ;
     length = lseek(  handle, 0L, SEEK_END ) ;

     close( handle ) ;

     printf( “The length of %s is %ld bytes \n”, name, length )
;
     }
```

8.3 Exercises

1. Write a program that determines the following statistics pertaining to a text file.

  i. Total number of characters

ii. Number of alphabetic characters
iii. Number of words
iv. Number of non alphabetic characters
v. Tabulates the usage of each letter of the alphabet.

2. Write a program that computes the value of Sin(x) for x in the range 0 to 2π in steps of 0.01 radians and stores them in a binary file. This look-up table is commonly used to improve program performance in practical programming rather than calculating values on the spot. Using the standard library random number generator to generate the angles compare the time it takes to 'calculate' Sin(x) for 100 values of x using the look-up table and calculating them straight. You might find the standard library time functions useful to compare times accurately.

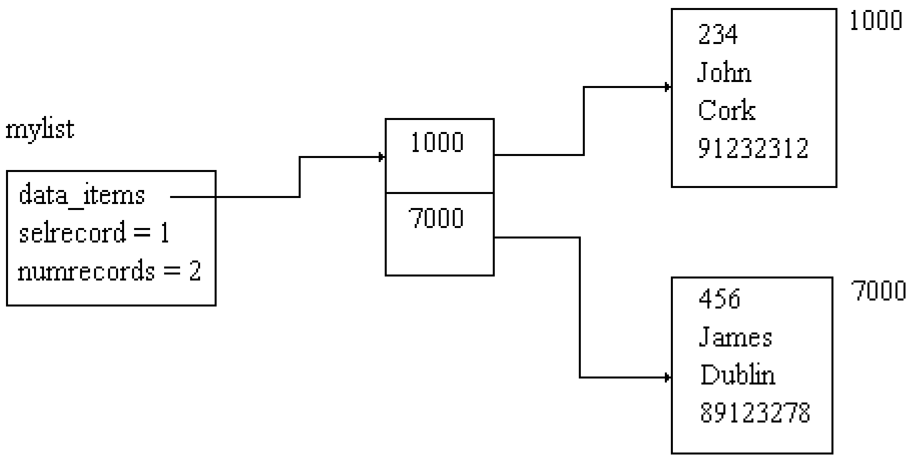
3. Programming Assignment : Simple DataBase.

Write a simple database program which stores and manages information of the type contained in the following structure by making use of a dynamically allocated array / list of structures of this type as described below.

```
typedef struct details {
                int rec_id ;
                char name[20] ;
                char address[80] ;
                long UCCid ;
                } DETAILS ;

typedef struct list {
                DETAILS **data_items ;
                int numrecords ;
                int selrecord ;
                } LIST ;
```

The list structure defined above contains three data items. <numrecords> is the total number of records in the list at present, <selrecord> is the current record selected, and <data_items> is a pointer to a pointer to type DETAILS, i.e. a doubly indirected pointer to the actual data.

The data is arranged as illustrated below in the case of a list with two records.



The list structure tells us that there are two records in the list, the current being the first in the list. The pointer mylist->data_items, of type DETAILS **, has been allocated memory to store two addresses, of type DETAILS *, i.e. the addresses for each individual record. Each of these individual pointers, i.e. *(mylist->data_items + i), has been allocated sufficient memory to store an individual record.

Your program should set up a data structure of the type described above and allow the user to perform the following tasks.

1. Add a record to the database.
2. Search for a record by field in the database.
3. Order the database by field.
4. Retrieve a record from the database.
5. Extract a record from the database, deleting it completely.
6. Save the database appropriately to a file.
7. Load an existing database from a file.

Your program should contain the following functions / features.

- `void initlist( LIST *list ) ;`
  This function should set <selrecord> = <numrecords> = 0 and <data_items> = NULL.

- `void add( LIST *list, DETAILS *new );`
  This function should add the record pointed to by <new> onto the end of the list pointed to by <list>. This means that <selrecord> and <numrecords> will have to be modified appropriately and the pointer
  <list->data_items> must be resized to hold the address of one extra record (using `realloc( )` for example), and memory must be allocated for the actual record i.e. `for *( list-> data_items + list->numrecords - 1 ).`

- `DETAILS *probe( LIST *list, int i ) ;`
  This function returns a pointer to the current record and automatically moves you onto the next record. If the current selection is 0, i.e. no record exists in the list, the function should return NULL. If the current selection is otherwise invalid <selrecord> should be reset to the first record and continue as normal. If <i> is equal to zero the list is to be reset and continue as normal, otherwise ignore <i>.

- `void extract( LIST *list );`
  This function removes the current selection completely from the list. It removes nothing if the current record is invalid.

- `void swap( LIST *list, int i, int j ) ;`
  Swaps records i and j in the list. Note you should only swap the actual addresses of the individual records.

- `void orderlist( LIST *list, int field ) ;`
  This function should ideally order the complete list in terms of the field given, e.g. in terms of name, UCCid, etc. However it will suffice to do this in terms of name only say.

- `DETAILS *search( LIST *list, char *item, int field ) ;`
  Search the list from the current position on, for the next occurrence of the searchitem, <item>, in a particular field of the list (with the same proviso as above). The function returns the null pointer if the item is not found, or a pointer to the particular record if it is found. The record becomes the current selection.

APPENDIX A : ASCII Character Set

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`
^A	1	01	☺	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	☻	STX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	ETX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOT	36	24	$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	◘	BS	40	28	(	72	48	H	104	68	h
^I	9	09	○	HT	41	29	)	73	49	I	105	69	i
^J	10	0A	◙	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	♪	CR	45	2D	–	77	4D	M	109	6D	m
^N	14	0E	♫	SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	☼	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	►	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◄	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	‼	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	¶	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	§	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	↨	ETB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SIB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C	∟	FS	60	3C	<	92	5C	\	124	7C	¦
^]	29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.