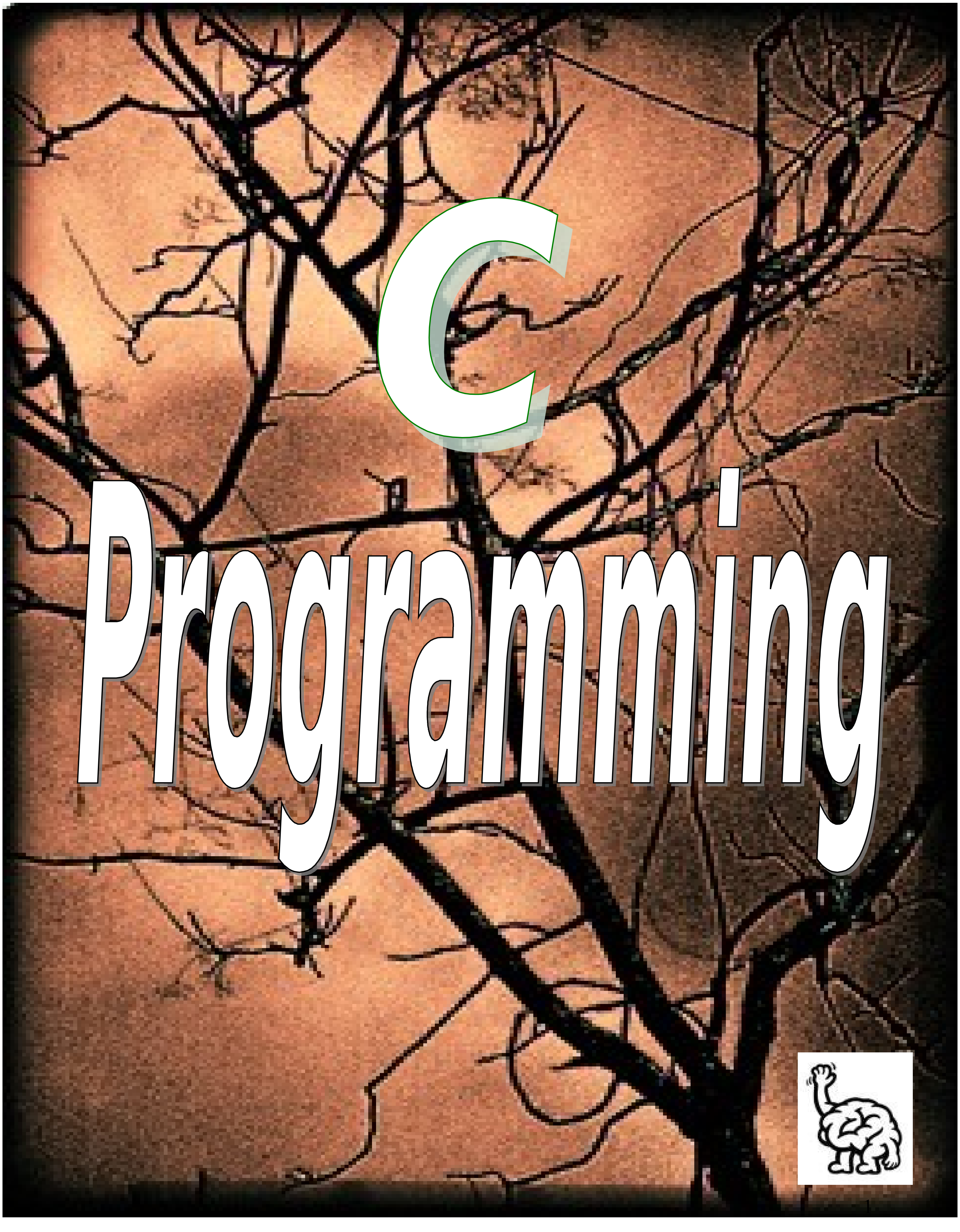
C Programming



C Programming

An Introduction

About C

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

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

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

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

Common C

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

ANSI C

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

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

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

A Simple Program

The following program is written in the C programming language.

```
#include <stdio.h>

main()
{
       printf("Programming in C is easy.\n");
}
```

A NOTE ABOUT C PROGRAMS
In C, lowercase and uppercase characters are very important! All commands in C must be lowercase. The C programs starting point is identified by the word

```
main()
```

This informs the computer as to where the program actually starts. The brackets that follow the keyword *main* indicate that there are no arguments supplied to this program (this will be examined later on).

The two braces, { and }, signify the begin and end segments of the program. The purpose of the statment

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

is to allow the use of the *printf* statement to provide program output. Text to be displayed by *printf()* must be enclosed in double quotes. The program has only one statement

```
printf("Programming in C is easy.\n");
```

printf() is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes "", it is printed without modification. There are some exceptions however. This has to do with the \ and % characters. These characters are modifier's, and for the present the \ followed by the n character represents a newline character. Thus the program prints

```
Programming in C is easy.
```

and the cursor is set to the beginning of the next line. As we shall see later on, what follows the \ character will determine what is printed, ie, a tab, clear screen, clear line etc. Another important thing to remember is that all C statements are terminated by a semi-colon ;

Summary of major points:

- program execution begins at *main()*

- keywords are written in lower-case
- statements are terminated with a semi-colon
- text strings are enclosed in double quotes
- C is case sensitive, use lower-case and try not to capitalise variable names
- \n means position the cursor on the beginning of the next line
- *printf()* can be used to display text to the screen
- The curly braces {} define the beginning and end of a program block.

# EXERCISE C1:
What will the following program output?

```
        #include <stdio.h>

        main()
        {
                printf("Programming in C is easy.\n");
                printf("And so is Pascal.\n");
        }
@ Programming in C is easy.
        And so is Pascal.
```

And this program?

```
        #include <stdio.h>

        main()
        {
                printf("The black dog was big. ");
                printf("The cow jumped over the moon.\n");
        }
@ The black dog was big. The cow jumped over the moon.
        _
```

Another thing about programming in C is that it is not necessary to repeatedly call the *printf* routine, so try and work out what the following program displays,

```
        #include <stdio.h>

        main()
        {
                printf("Hello...\n..oh my\n...when do i stop?\n");
        }

@ Hello...
        ..oh my
        ...when do i stop?
        _
```

VARIABLES

C provides the programmer with FOUR basic data types. User defined variables must be declared before they can be used in a program.

Get into the habit of declaring variables using lowercase characters. Remember that C is case sensitive, so even though the two variables listed below have the same name, they are considered different variables in C.

```
sum
Sum
```

The declaration of variables is done after the opening brace of *main()*,

```
#include <stdio.h>

main()
{
        int sum;

        sum = 500 + 15;
        printf("The sum of 500 and 15 is %d\n", sum);
}
```

It is possible to declare variables elsewhere in a program, but lets start simply and then get into variations later on.

The basic format for declaring variables is

```
data_type   var, var, ... ;
```

where *data_type* is one of the four basic types, an *integer*, *character*, *float*, or *double* type.

The program declares the variable *sum* to be of type INTEGER (int). The variable *sum* is then assigned the value of 500 + 15 by using the assignment operator, the = sign.

```
sum = 500 + 15;
```

Now lets look more closely at the *printf()* statement. It has two arguments, separated by a comma. Lets look at the first argument,

```
"The sum of 500 and 15 is %d\n"
```

The *%* sign is a special character in C. It is used to display the value of variables. When the program is executed, C starts printing the text until it finds a *%* character. If it finds one, it looks up for the next argument (in this case *sum*), displays its value, then continues on.

The *d* character that follows the *%* indicates that a decimal integer is expected. So, when the *%d* sign is reached, the next argument to the *printf()* routine is looked up (in this case the variable *sum*, which is 515), and displayed. The *\n* is then executed which prints the newline character.

The output of the program is thus,

```
The sum of 500 and 15 is 515
_
```

Some of the formatters for *printf* are,

Cursor Control Formatters

\n	newline
\t	tab
\r	carriage return
\f	form feed
\v	vertical tab

Variable Formatters

%d	decimal integer
%c	character
%s	string or character array
%f	float
%e	double

The following program prints out two integer values separated by a TAB
It does this by using the \t cursor control formatter

```
#include <stdio.h>

main()
{
        int sum, value;

        sum = 10;
        value = 15;
        printf("%d\t%d\n", sum, value);
}
```

Program output looks like

```
10      15
_
```

# EXERCISE C2:
What is the output of the following program?

```
#include <stdio.h>

main()
{
        int   value1, value2, sum;

        value1 = 35;
        value2 = 18;
        sum = value1 + value2;
```

```
                printf("The sum of %d and %d is %d\n", value1, value2, sum);
        }
```

```
@ The sum of 35 and 18 is 53
        _
```

Note that the program declares three variables, all integers, on the same declaration line. This could've been done by three separate declarations,

```
        int  value1;
        int  value2;
        int  sum;
```

COMMENTS
The addition of comments inside programs is desirable. These may be added to C programs by enclosing them as follows,

```
        /* bla bla bla bla bla bla */
```

Note that the **/*** opens the comment field and ***/** closes the comment field. Comments may span multiple lines. Comments may not be nested one inside another.

```
        /* this is a comment. /* this comment is inside */ wrong */
```

In the above example, the first occurrence of ***/** closes the comment statement for the entire line, meaning that the text ***wrong*** is interpreted as a C statement or variable, and in this example, generates an error.

What Comments Are Used For

- documentation of variables and their usage
- explaining difficult sections of code
- describes the program, author, date, modification changes, revisions etc
- copyrighting

Basic Structure of C Programs
C programs are essentially constructed in the following manner, as a number of well defined sections.

```
        /* HEADER SECTION                        */
        /* Contains name, author, revision number*/

        /* INCLUDE SECTION                       */
        /* contains #include statements          */

        /* CONSTANTS AND TYPES SECTION           */
        /* contains types and #defines           */

        /* GLOBAL VARIABLES SECTION              */
        /* any global variables declared here    */
```

```
/* FUNCTIONS SECTION                          */
/* user defined functions                     */

/* main() SECTION                             */

int main()
{

}
```

Adhering to a well defined structured layout will make your programs

- easy to read
- easy to modify
- consistent in format
- self documenting

MORE ABOUT VARIABLES

Variables must begin with a character or underscore, and may be followed by any combination of characters, underscores, or the digits 0 - 9. The following is a list of valid variable names,

```
summary
exit_flag
i
Jerry7
Number_of_moves
_valid_flag
```

You should ensure that you use meaningful names for your variables. The reasons for this are,

- meaningful names for variables are self documenting (see what they do at a glance)
- they are easier to understand
- there is no correlation with the amount of space used in the .EXE file
- makes programs easier to read

# EXERCISE C3:

Why are the variables in the following list invalid,

```
value$sum
exit flag
3lotsofmoney
char
```

```
@ value$sum     contains a $
        exit flag       contains a space
        3lotsofmoney    begins with a digit
        char            is a reserved keyword
```

VARIABLE NAMES AND PREFIXES WHEN WRITING WINDOWS OR OS/2 PROGRAMS

During the development of OS/2, it became common to add prefix letters to variable names to indicate the data type of variables.

This enabled programmers to identify the data type of the variable without looking at its declaration, thus they could easily check to see if they were performing the correct operations on the data type and hopefully, reduce the number of errors.

Prefix	**Purpose or Type**
b	a byte value
c	count or size
clr	a variable that holds a color
f	bitfields or flags
h	a handle
hwnd	a window handle
id	an identity
l	a long integer
msg	a message
P	a Pointer
rc	return value
s	short integer
ul	unsigned long integer
us	unsigned short integer
sz	a null terminated string variable
psz	a pointer to a null terminated string variable

DATA TYPES AND CONSTANTS

The four basic data types are

- **INTEGER**
  These are whole numbers, both positive and negative. Unsigned integers (positive values only) are supported. In addition, there are short and long integers.

  The keyword used to define integers is,

  ```
          int
  ```

  An example of an integer value is 32. An example of declaring an integer variable called **sum** is,

  ```
          int sum;
  sum = 20;
  ```

- **FLOATING POINT**
  These are numbers which contain fractional parts, both positive and negative. The keyword used to define float variables is,

-
  ```
        float
  ```

  An example of a float value is 34.12. An example of declaring a float variable called **money** is,

  ```
        float money;
  money = 0.12;
  ```

- **DOUBLE**
  These are exponetional numbers, both positive and negative. The keyword used to define double variables is,

-
  ```
        double
  ```

  An example of a double value is 3.0E2. An example of declaring a double variable called **big** is,

  ```
        double big;
  big = 312E+7;
  ```

- **CHARACTER**
  These are single characters. The keyword used to define character variables is,
-

-
  ```
        char
  ```

  An example of a character value is the letter **A**. An example of declaring a character variable called **letter** is,

  ```
        char letter;
  letter = 'A';
  ```

  Note the assignment of the character *A* to the variable *letter* is done by enclosing the value in **single quotes**. Remember the golden rule: Single character - Use single quotes.

Sample program illustrating each data type

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

```
main()
{
        int   sum;
        float money;
        char  letter;
        double pi;

        sum = 10;               /* assign integer value */
        money = 2.21;           /* assign float value */
        letter = 'A';           /* assign character value */
        pi = 2.01E6;            /* assign a double value */

        printf("value of sum = %d\n", sum );
        printf("value of money = %f\n", money );
        printf("value of letter = %c\n", letter );
        printf("value of pi = %e\n", pi );
}
```

Sample program output

```
value of sum = 10
value of money = 2.210000
value of letter = A
value of pi = 2.010000e+06
```

INITIALISING DATA VARIABLES AT DECLARATION TIME

Unlike PASCAL, in C variables may be initialised with a value when they are declared. Consider the following declaration, which declares an integer variable *count* which is initialised to 10.

```
int  count = 10;
```

SIMPLE ASSIGNMENT OF VALUES TO VARIABLES

The = operator is used to assign values to data variables. Consider the following statement, which assigns the value 32 an integer variable *count*, and the letter **A** to the character variable *letter*

```
count = 32;
letter = 'A';
```

THE VALUE OF VARIABLES AT DECLARATION TIME

Lets examine what the default value a variable is assigned when its declared. To do this, lets consider the following program, which declares two variables, *count* which is an integer, and *letter* which is a character.

Neither variable is pre-initialised. The value of each variable is printed out using a *printf()* statement.

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

```
main()
{
        int   count;
        char  letter;

        printf("Count = %d\n", count);
        printf("Letter = %c\n", letter);
}
```

```
Sample program output
Count = 26494
Letter = f
```

It can be seen from the sample output that the values which each of the variables take on at declaration time are **no-zero**. In C, this is common, and programmers must ensure that variables are assigned values before using them.

If the program was run again, the output could well have different values for each of the variables. We can never assume that variables declare in the manner above will take on a specific value.

Some compilers may issue warnings related to the use of variables, and Turbo C from Borland issues the following warning,

```
possible use of 'count' before definition in function main
```

RADIX CHANGING

Data numbers may be expressed in any base by simply altering the modifier, eg, decimal, octal, or hexadecimal. This is achieved by the letter which follows the % sign related to the *printf* argument.

```
#include <stdio.h>

main() /* Prints the same value in Decimal, Hex and Octal */
{
        int   number = 100;

        printf("In decimal the number is %d\n", number);
        printf("In hex the number is %x\n", number);
        printf("In octal the number is %o\n", number);
        /* what about %X\n as an argument?  */
}
```

```
Sample program output
In decimal the number is 100
In hex the number is 64
In octal the number is 144
```

Note how the variable *number* is initialised to zero at the time of its declaration.

DEFINING VARIABLES IN OCTAL AND HEXADECIMAL

Often, when writing systems programs, the programmer needs to use a different number base rather than the default decimal.

Integer constants can be defined in octal or hex by using the associated prefix, eg, to define an integer as an octal constant use *%o*

```
        int    sum = %o567;
```

To define an integer as a hex constant use *%0x*

```
        int    sum = %0x7ab4;
        int    flag = %0x7AB4;     /* Note upper or lowercase hex ok */
```

MORE ABOUT FLOAT AND DOUBLE VARIABLES

C displays both float and double variables to six decimal places. This does NOT refer to the precision (accuracy) of which the number is actually stored, only how many decimal places *printf()* uses to display these variable types.

The following program illustrates how the different data types are declared and displayed,

```
        #include <stdio.h>

        main()
        {
                int     sum = 100;
                char    letter = 'Z';
                float   set1 = 23.567;
                double  num2 = 11e+23;

                printf("Integer variable is %d\n", sum);
                printf("Character is %c\n", letter);
                printf("Float variable is %f\n", set1);
                printf("Double variable is %e\n", num2);
        }
```

```
        Sample program output
        Integer variable is 100
        Character variable is Z
        Float variable is 23.567000
        Double variable is 11.000000e23
```

To change the number of decimal places printed out for float or double variables, modify the %f or %e to include a precision value, eg,

```
        printf("Float variable is %.2f\n", set1 );
```

In this case, the use of %.2f limits the output to two decimal places, and the output now looks like

```
        Sample program output
        Integer variable is 100
        Character variable is Z
```

```
Float variable is 23.56
Double variable is 11.000000e23
```

SPECIAL NOTE ABOUT DATA TYPE CONVERSION

Consider the following program,

```
        #include <stdio.h>

        main()
        {
                int  value1 = 12, value2 = 5;
                float answer = 0;

                answer = value1 / value2;
                printf("The value of %d divided by %d is %f\n",value1,value2,answer
);
        }
```

Sample program output

```
The value of 12 divided by 5 is 2.000000
```

Even though the above declaration seems to work, it is not always 100% reliable. **Note** how *answer* does not contain a proper fractional part (ie, all zero's).

To ensure that the correct result always occurs, the data type of *value1* and *value2* should be converted to a float type before assigning to the float variable *answer*. The following change illustrates how this can be done,

```
answer = (float)value1 / (float)value2;
```

DIFFERENT TYPES OF INTEGERS

A normal integer is limited in range to +-32767. This value differs from computer to computer. It is possible in C to specify that an integer be stored in four memory locations instead of the normal two. This increases the effective range and allows very large integers to be stored. The way in which this is done is as follows,

```
long int big_number = 245032L;
```

To display a long integer, use %l, ie,

```
printf("A larger number is %l\n", big_number );
```

Short integers are also available, eg,

```
short int   small_value = 114h;
printf("The value is %h\n", small_value);
```

Unsigned integers (positive values only) can also be defined.

The size occupied by integers varies upon the machine hardware. ANSI C (American National Standards Institute) has tried to standardise upon the size of data types, and hence the number range of each type.

The following information is from the on-line help of the Turbo C compiler,

```
        Type: int
          Integer data type

        Variables of type int are one word in length.
        They can be signed (default) or unsigned,
        which means they have a range of -32768 to
        32767 and 0 to 65535, respectively.


        Type modifiers: signed, unsigned, short, long

        A type modifier alters the meaning of the base
        type to yield a new type. Each of the above
        can be applied to the base type int. The
        modifiers signed and unsigned can be applied
        to the base type char. In addition, long can
        be applied to double. When the base type is
        ommitted from a declaration, int is assumed.

        Examples:
        long              x;  /* int is implied */
        unsigned char     ch;
        signed int        i;  /* signed is default */
        unsigned long int l;  /* int ok, not needed */
```

PREPROCESSOR STATEMENTS

The **define** statement is used to make programs more readable. Consider the following examples,

```
        #define TRUE    1    /* Don't use a semi-colon , # must be first character
on line */
        #define FALSE   0
        #define NULL    0
        #define AND     &
        #define OR      |
        #define EQUALS  ==

        game_over = TRUE;
        while( list_pointer != NULL )
                ................
```

Note that preprocessor statements begin with a # symbol, and are NOT terminated by a semi-colon. Traditionally, preprocessor statements are listed at the beginning of the source file.

Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like TRUE) which occur in the C program are replaced by their value (like 1). Once this substitution has taken place by the preprocessor, the program is then compiled.

In general, preprocessor constants are written in **UPPERCASE**.

Click here for more information of preprocessor statements, including macros.

# Exercise C4:
Use pre-processor statements to replace the following constants

```
0.312
W
37
```

@ Use pre-processor statements to replace the following constants

```
0.312
W
37
```

```
#define smallvalue  0.312
#define letter      'W'
#define smallint    37
```

LITERAL SUBSTITUTION OF SYMBOLIC CONSTANTS USING #define
Lets now examine a few examples of using these symbolic constants in our programs. Consider the following program which defines a constant called TAX_RATE.

```
#include <stdio.h>

#define TAX_RATE  0.10

main()
{
        float balance;
        float tax;

        balance = 72.10;
        tax = balance * TAX_RATE;
        printf("The tax on %.2f is %.2f\n", balance, tax );
}
```

The pre-processor first replaces all symbolic constants before the program is compiled, so after preprocessing the file (and before its compiled), it now looks like,

```
#include <stdio.h>

#define TAX_RATE  0.10
```

```
main()
{
        float balance;
        float tax;

        balance = 72.10;
        tax = balance * 0.10;
        printf("The tax on %.2f is %.2f\n", balance, tax );
}
```

YOU CANNOT ASSIGN VALUES TO THE SYMBOLIC CONSTANTS

Considering the above program as an example, look at the changes we have made below. We have added a statement which tries to change the TAX_RATE to a new value.

```
#include <stdio.h>

#define TAX_RATE 0.10

main ()
{
        float balance;
        float tax;

        balance = 72.10;
        TAX_RATE = 0.15;
        tax = balance * TAX_RATE;
        printf("The tax on %.2f is %.2f\n", balance, tax );
}
```

This is **illegal**. You cannot re-assign a new value to a symbolic constant.

ITS LITERAL SUBSTITUTION, SO BEWARE OF ERRORS

As shown above, the preprocessor performs literal substitution of symbolic constants. Lets modify the previous program slightly, and introduce an error to highlight a problem.

```
#include <stdio.h>

#define TAX_RATE  0.10;

main()
{
        float balance;
        float tax;

        balance = 72.10;
        tax = (balance * TAX_RATE )+ 10.02;
        printf("The tax on %.2f is %.2f\n", balance, tax );
}
```

In this case, the error that has been introduced is that the *#define* is terminated with a semi-colon. The preprocessor performs the substitution and the offending line (which is flagged as an error by the compiler) looks like

```
                tax = (balance * 0.10; )+ 10.02;
```
However, you do not see the output of the preprocessor. If you are using TURBO C, you will only see
```
                tax = (balance * TAX_RATE )+ 10.02;
```
flagged as an error, and this actually looks okay (but its not! after substitution takes place).

MAKING PROGRAMS EASY TO MAINTAIN BY USING #define
The whole point of using *#define* in your programs is to make them easier to read and modify. Considering the above programs as examples, what changes would you need to make if the TAX_RATE was changed to 20%.

Obviously, the answer is once, where the *#define* statement which declares the symbolic constant and its value occurs. You would change it to read

```
        #define TAX_RATE = 0.20
```

Without the use of symbolic constants, you would hard code the value 0.20 in your program, and this might occur several times (or tens of times).

This would make changes difficult, because you would need to search and replace every occurrence in the program. However, as the programs get larger, **what would happen if you actually used the value 0.20 in a calculation that had nothing to do with the TAX_RATE!**

SUMMARY OF #define
- allow the use of symbolic constants in programs
- in general, symbols are written in uppercase
- are not terminated with a semi-colon
- generally occur at the beginning of the file
- each occurrence of the symbol is replaced by its value
- makes programs readable and easy to maintain

HEADER FILES
Header files contain definitions of functions and variables which can be incorporated into any C program by using the pre-processor *#include* statement. Standard header files are provided with each compiler, and cover a range of areas, string handling, mathematical, data conversion, printing and reading of variables.

To use any of the standard functions, the appropriate header file should be included. This is done at the beginning of the C source file. For example, to use the function *printf()* in a program, the line

```
        #include  <stdio.h>
```
should be at the beginning of the source file, because the definition for *printf()* is found in the file *stdio.h* All header files have the extension .h and generally reside in the /include subdirectory.

```
        #include <stdio.h>
        #include "mydecls.h"
The use of angle brackets <> informs the compiler to search the compilers include
directory for the specified file. The use of the double quotes "" around the
filename inform the compiler to search in the current directory for the specified
file.
```

Practise Exercise 1: Defining Variables

1. Declare an integer called sum

2. Declare a character called letter

3. Define a constant called TRUE which has a value of 1

4. Declare a variable called money which can be used to hold currency

5. Declare a variable called arctan which will hold scientific notation values (+e)

6. Declare an integer variable called total and initialise it to zero.

7. Declare a variable called loop, which can hold an integer value.

8. Define a constant called GST with a value of .125

Answers to Practise Exercise 1: Defining Variables

1. Declare an integer called sum

```
        int sum;
```

2. Declare a character called letter

```
        char letter;
```

3. Define a constant called TRUE which has a value of 1

```
        #define TRUE 1
```

4. Declare a variable called money which can be used to hold currency

float money;

5. Declare a variable called arctan which will hold scientific notation values (+e)

```
double arctan;
```

6. Declare an integer variable called total and initialise it to zero.

```
int total;
total = 0;
```

7. Declare a variable called loop, which can hold an integer value.

```
int loop;
```

8. Define a constant called GST with a value of .125

```
#define GST 0.125
```

ARITHMETIC OPERATORS
The symbols of the arithmetic operators are:-

Operation	**Operator**	**Comment**	**Value of Sum before**	**Value of sum after**
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6
Subtraction	-	sum = sum -2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

The following code fragment adds the variables *loop* and *count* together, leaving the result in the variable *sum*

```
sum = loop + count;
```

Note: If the modulus **%** sign is needed to be displayed as part of a text string, use two, ie %%

```
        #include <stdio.h>

        main()
        {
                int sum = 50;
                float modulus;

                modulus = sum % 10;
                printf("The %% of %d by 10 is %f\n", sum, modulus);
        }
```

# EXERCISE C5:

What does the following change do to the printed output of the previous program?

```
        printf("The %% of %d by 10 is %.2f\n", sum, modulus);
```

```
@       #include <stdio.h>

        main()
        {
                int sum = 50;
                float modulus;

                modulus = sum % 10;
                printf("The %% of %d by 10 is %.2f\n", sum, modulus);
        }


        The % of 50 by 10 is 0.00
        _
```

Practise Exercise 2: Assignments

1. Assign the value of the variable number1 to the variable total

2. Assign the sum of the two variables loop_count and petrol_cost to the variable sum

3. Divide the variable total by the value 10 and leave the result in the variable discount

4. Assign the character W to the char variable letter

5. Assign the decimal result of dividing the integer variable sum by 3 into the float variable costing. Use type casting to ensure that the remainder is also held by the float variable.

Answers: Practise Exercise 2: Assignments

1. Assign the value of the variable number1 to the variable total

```
        total = number1;
```

2. Assign the sum of the two variables loop_count and petrol_cost to the variable sum

```
        sum = loop_count + petrol_cost;
```

3. Divide the variable total by the value 10 and leave the result in the variable discount

```
        discount = total / 10;
```

4. Assign the character W to the char variable letter

```
        letter = 'W';
```

5. Assign the decimal result of dividing the integer variable sum by 3 into the float variable costing. Use type casting to ensure that the remainder is also held by the float variable.

```
        costing = (float) sum / 3;
```

PRE/POST INCREMENT/DECREMENT OPERATORS
PRE means do the operation first followed by any assignment operation. POST means do the operation after any assignment operation. Consider the following statements

```
        ++count;         /* PRE Increment, means add one to count */
        count++;         /* POST Increment, means add one to count */
```

In the above example, because the value of *count* is not assigned to any variable, the effects of the PRE/POST operation are not clearly visible.

Lets examine what happens when we use the operator along with an assignment operation. Consider the following program,

```
        #include <stdio.h>

        main()
        {
                int count = 0, loop;

                loop = ++count;  /* same as count = count + 1; loop = count;  */
                printf("loop = %d, count = %d\n", loop, count);

                loop = count++;  /* same as loop = count;  count = count + 1;  */
                printf("loop = %d, count = %d\n", loop, count);
        }
```

If the operator precedes (is on the left hand side) of the variable, the operation is performed first, so the statement

```
        loop = ++count;
```

really means increment *count* first, then assign the new value of *count* to *loop*.

Which way do you write it?
Where the increment/decrement operation is used to adjust the value of a variable, and is not involved in an assignment operation, which should you use,

```
        ++loop_count;
or
        loop_count++;
```

The answer is, it really does not matter. It does seem that there is a preference amongst C programmers to use the post form.

Something to watch out for
Whilst we are on the subject, do not get into the habit of using a space(s) between the variable name and the pre/post operator.

```
        loop_count ++;
```

Try to be explicit in *binding* the operator tightly by leaving no gap.

GOOD FORM
Perhaps we should say *programming style* or *readability*. The most common complaints we would have about beginning C programmers can be summarised as,

- they have poor layout
- their programs are hard to read

Your programs will be quicker to write and easier to debug if you get into the habit of actually formatting the layout correctly as you write it.

For instance, look at the program below

```
#include<stdio.h>
main()
   {
    int sum,loop,kettle,job;
    char Whoknows;

         sum=9;
   loop=7;
  whoKnows='A';
printf("Whoknows=%c,Kettle=%d\n",whoknows,kettle);
}
```

It is our contention that the program is hard to read, and because of this, will be difficult to debug for errors by an inexperienced programmer. It also contains a few deliberate mistakes!

Okay then, lets rewrite the program using good form.

```
#include <stdio.h>

main()
{
        int sum, loop, kettle, job;
        char whoknows;

        sum = 9;
        loop = 7;
        whoknows = 'A';
        printf( "Whoknows = %c, Kettle = %d\n", whoknows, kettle );
}
```

We have also corrected the mistakes. The major differences are

- the { and } braces directly line up underneath each other
  This allows us to check ident levels and ensure that statements belong to the correct block of code. This becomes vital as programs become more complex
- spaces are inserted for readability
  We as humans write sentences using spaces between words. This helps our comprehension of what we read (if you dont believe me, try reading the following sentence. wishihadadollarforeverytimeimadeamistake. The insertion of spaces will also help us identify mistakes quicker.
- good indentation
  Indent levels (tab stops) are clearly used to block statements, here we clearly see and identify functions, and the statements which belong to each { } program body.

Programs to help you
There are a number of shareware programs available in the public domain which will assist you in the areas of available to [registered users only]

- checking for code correctness
- converting between tabs and spaces

- formatting the layout with indentation etc
- building a tree representation of program flow
- generating cross references
- checking syntax

Please note that the above are all MSDOS based programs. Perhaps the most famous of all C program utilities is *lint*.

KEYBOARD INPUT

There is a function in C which allows the programmer to accept input from a keyboard. The following program illustrates the use of this function,

```
#include <stdio.h>

main()     /* program which introduces keyboard input */
{
        int  number;

        printf("Type in a number \n");
        scanf("%d", &number);
        printf("The number you typed was %d\n", number);
}
```

An integer called *number* is defined. A prompt to enter in a number is then printed using the statement

```
printf("Type in a number \n:");
```

The *scanf* routine, which accepts the response, has two arguments. The first ("%d") specifies what type of data type is expected (ie char, int, or float).

The second argument (&number) specifies the variable into which the typed response will be placed. In this case the response will be placed into the memory location associated with the variable *number*.

This explains the special significance of the & character (which means the address of).

Sample program illustrating use of scanf() to read integers, characters and floats

```
#include < stdio.h >

main()
{
        int sum;
        char letter;
        float money;

        printf("Please enter an integer value ");
        scanf("%d", &sum );
```

```
		printf("Please enter a character ");
		/* the leading space before the %c ignores space characters in the
input */
		scanf("  %c", &letter );

		printf("Please enter a float variable ");
		scanf("%f", &money );

		printf("\nThe variables you entered were\n");
		printf("value of sum = %d\n", sum );
		printf("value of letter = %c\n", letter );
		printf("value of money = %f\n", money );
	}
```

This program illustrates several important points.

- the c language provides no error checking for user input. The user is expected to enter the correct data type. For instance, if a user entered a character when an integer value was expected, the program may enter an infinite loop or abort abnormally.
- its up to the programmer to **validate** data for correct type and range of values.

Practise Exercise 3: printf() and scanf()

1. Use a printf statement to print out the value of the integer variable sum

2. Use a printf statement to print out the text string "Welcome", followed by a newline.

3. Use a printf statement to print out the character variable letter

4. Use a printf statement to print out the float variable discount

5. Use a printf statement to print out the float variable dump using two decimal places

6. Use a scanf statement to read a decimal value from the keyboard, into the integer variable sum

7. Use a scanf statement to read a float variable into the variable discount_rate

8. Use a scanf statement to read a single character from the keyboard into the variable operator. Skip leading blanks, tabs and newline characters.

Answers: Practise Exercise 3: printf() and scanf()

1. Use a printf statement to print out the value of the integer variable sum

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

2. Use a printf statement to print out the text string "Welcome", followed by a newline.

```
        printf("Welcome\n");
```

3. Use a printf statement to print out the character variable letter

```
        printf("%c", letter);
```

4. Use a printf statement to print out the float variable discount

```
        printf("%f", discount);
```

5. Use a printf statement to print out the float variable dump using two decimal places

```
        printf("%.2f", dump);
```

6. Use a scanf statement to read a decimal value from the keyboard, into the integer variable sum

```
        scanf("%d", &sum);
```

7. Use a scanf statement to read a float variable into the variable discount_rate

```
        scanf("%f", &discount_rate);
```

8. Use a scanf statement to read a single character from the keyboard into the variable operator. Skip leading blanks, tabs and newline characters.

```
        scanf(" %c", &operator);
```

THE RELATIONAL OPERATORS
These allow the comparision of two or more variables.

```
        ==       equal to
        !=       not equal
        <        less than
        <=       less than or equal to
        >        greater than
        >=       greater than or equal to
```

In the next few screens, these will be used in *for* loops and *if* statements.

The operator

```
        <>
```

may be legal in Pascal, **but is illegal in C.**

ITERATION, FOR LOOPS

The basic format of the for statement is,

```
        for( start condition; continue condition; re-evaulation )
                program statement;
```

```
        /* sample program using a for statement */
        #include <stdio.h>

        main()   /* Program introduces the for statement, counts to ten */
        {
                int  count;

                for( count = 1; count <= 10; count = count + 1 )
                        printf("%d ", count );

                printf("\n");
        }
```

The program declares an integer variable *count*. The first part of the *for* statement

```
        for( count = 1;
```

initialises the value of *count* to 1. The *for* loop continues whilst the condition

```
        count <= 10;
```

evaluates as TRUE. As the variable *count* has just been initialised to 1, this condition is TRUE and so the program statement

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

is executed, which prints the value of *count* to the screen, followed by a space character.

Next, the remaining statement of the *for* is executed

```
        count = count + 1 );
```
which adds one to the current value of *count*. Control now passes back to the conditional test,

```
        count <= 10;
```
which evaluates as true, so the program statement

```
                printf("%d ", count );
```
is executed. *Count* is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test

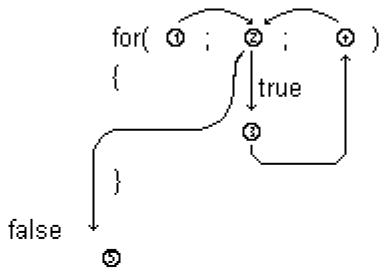
```
        count <= 10;
```
evaluates as FALSE, and the *for* loop terminates, and program control passes to the statement

```
        printf("\n");
```
which prints a newline, and then the program terminates, as there are no more statements left to execute.

```
        /* sample program using a for statement */
        #include <stdio.h>

        main()
        {
                int  n, t_number;

                t_number = 0;
                for( n = 1; n <= 200; n = n + 1 )
                        t_number = t_number + n;

                printf("The 200th triangular_number is %d\n", t_number);
        }
```

The above program uses a *for* loop to calculate the sum of the numbers from 1 to 200 inclusive (said to be the *triangular number*).

The following diagram shows the order of processing each part of a *for*



An example of using a for loop to print out characters

```
#include <stdio.h>

main()
{
	char letter;
	for( letter = 'A'; letter <= 'E'; letter = letter + 1 ) {
		printf("%c ", letter);
	}
}


Sample Program Output
A B C D E
```

An example of using a for loop to count numbers, using two initialisations

```
#include <stdio.h>

main()
{
	int total, loop;
	for( total = 0, loop = 1; loop <= 10; loop = loop + 1 ){
		total = total + loop;
	}
	printf("Total = %d\n", total );
}
```

In the above example, the variable *total* is initialised to 0 as the first part of the for loop. The two statements,

```
	for( total = 0, loop = 1;
```

are part of the initialisation. This illustrates that more than one statement is allowed, as long as they are separated by **commas**.

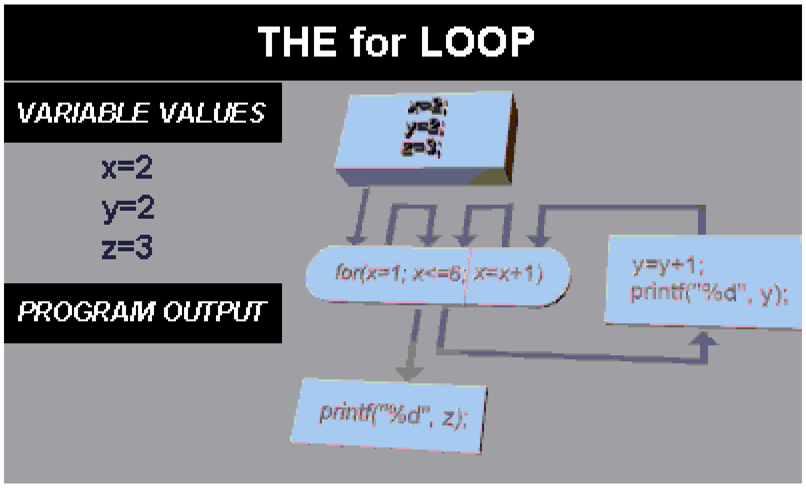
Graphical Animation of *for* loop
To demonstrate the operation of the *for* statement, lets consider a series of animations.

The code we will be using is

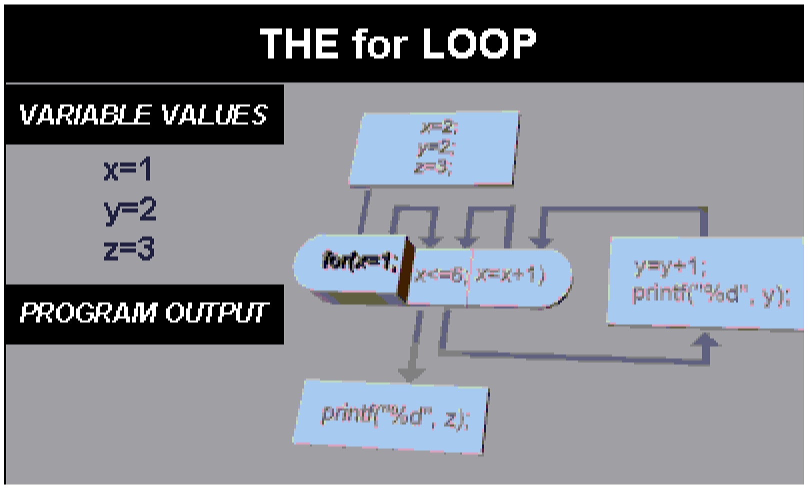
```
#include <stdio.h>

main() {
	int x, y, z;

	x = 2;
	y = 2;
	z = 3;

	for( x = 1; x <= 6; x = x + 1 ) {
		printf("%d", y );
		y = y + 1;
```

```
            }
            printf("%d", z );
    }
```

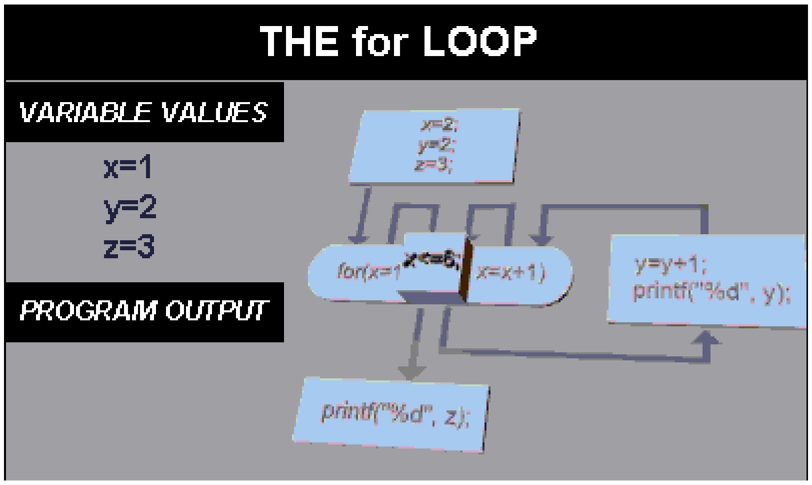
The following diagram shows the initial state of the program, after the initialisation of the variables *x, y,* and *z*.



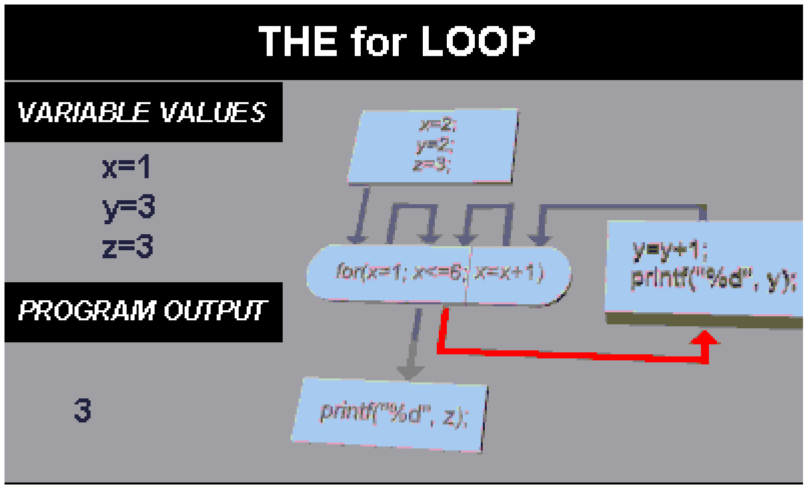
On entry to the *for* statement, the first expression is executed, which in our example assigns the value 1 to *x*. This can be seen in the graphic shown below (Note: see the Variable Values: section)



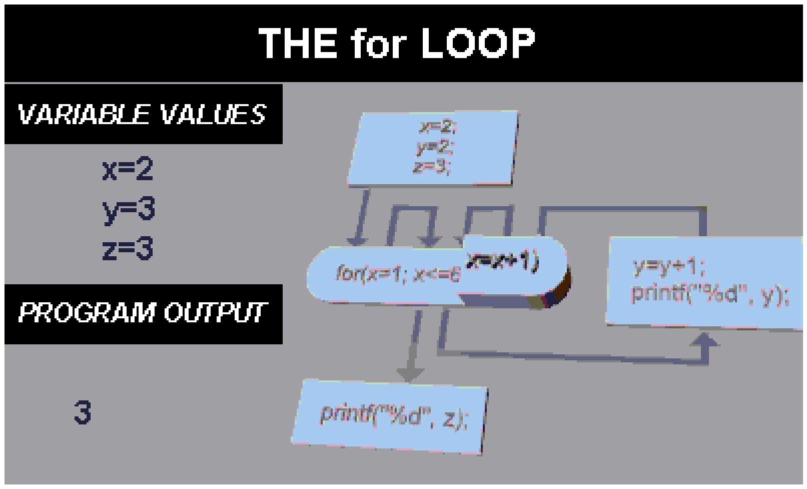
The next part of the *for* is executed, which tests the value of the loop variable *x* against the constant **6**.



It can be seen from the variable window that *x* has a current value of 1, so the test is successful, and program flow branches to execute the statements of the *for body*, which prints out the value of *y*, then adds 1 to *y*. You can see the program output and the state of the variables shown in the graphic below.

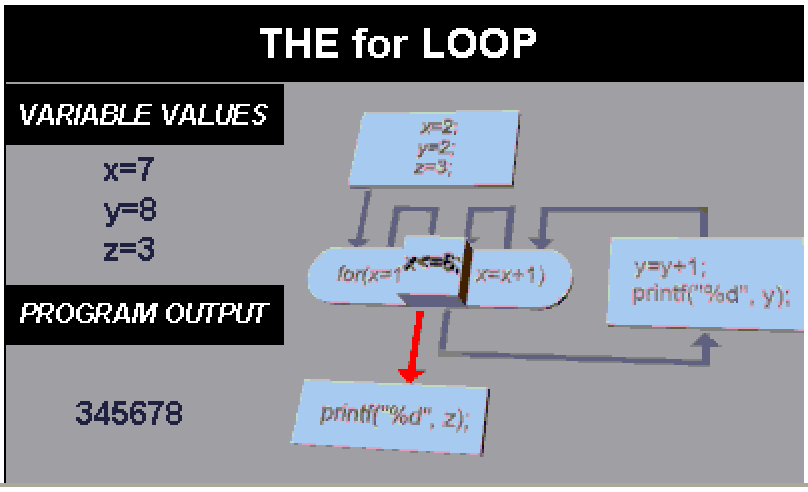


After executing the statements of the *for body*, execution returns to the last part of the *for* statement. Here, the value of *x* is incremented by 1. This is seen by the value of *x* changing to 2.



Next, the condition of the *for* variable is tested again. It continues because the value of it (2) is less than 6, so the body of the loop is executed again.

Execution continues till the value of *x* reaches 7. Lets now jump ahead in the animation to see this. Here, the condition test will fail, and the *for* statement finishes, passing control to the statement which follows.



# EXERCISE C6:
Rewrite the previous program by calculating the 200th triangular number, and make the program shorter (if possible).

```
@       #include <stdio.h>

        main()
        {
                int  n = 1, t_number = 0;

                for( ; n <= 200; n++ )
                t_number = t_number + n;

        printf("The 200th triangular_number is %d\n", t_number);
}
```

CLASS EXERCISE C7
What is the difference between the two statements,

```
a == 2
a = 2
```

```
@       a == 2          equality test
        a = 2           assignment
```

CLASS EXERCISE C8
Change the printf line of the above program to the following,

```
printf(" %2d           %2d\n",n,t_number);
```

What does the inclusion of the 2 in the %d statements achieve?

@ The inclusion of the 2 in the %d statements achieves a field width of two places, and prints a leading 0 where the value is less than 10.

EXERCISE C9
Create a C program which calculates the triangular number of the users request, read from the keyboard using **scanf()**. A triangular number is the sum of the preceding numbers, so the triangular number 7 has a value of

7 + 6 + 5 + 4 + 3 + 2 + 1

```
@       #include <stdio.h>

        main()
        {
                int  n = 1, t_number = 0, input;

                printf("Enter a number\n");
                scanf("%d", &input);
                for( ; n <= input; n++ )
                        t_number = t_number + n;

                printf("The triangular_number of %d is %d\n", input, t_number);
        }
```

Practise Exercise 4: for loops

1. Write a for loop to print out the values 1 to 10 on separate lines.

```
for( loop = 1; loop <= 10; loop = loop + 1 )
        printf("%d\n", loop) ;
```

2. Write a for loop which will produce the following output (hint: use two nested for loops)

```
1
22
333
4444
55555

for( loop = 1; loop <= 5; loop = loop + 1 )
{
        for( count = 1; count <= loop; count  = count + 1 )
                printf("%d", count );
        printf("\n");
}
```

3. Write a for loop which sums all values between 10 and 100 into a variable called *total*. Assume that *total* has NOT been initialised to zero.

```
for( loop = 10, total = 0; loop <= 100; loop = loop + 1 )
        total = total + loop;
```

4. Write a for loop to print out the character set from A-Z.

```
for( ch = 'A'; ch <= 'Z'; ch = ch + 1 )
        printf("%c", ch );
printf("\n");
```

THE WHILE STATEMENT
The *while* provides a mechanism for repeating C statements whilst a condition is true. Its format is,

```
while( condition )
        program statement;
```

Somewhere within the body of the *while* loop a statement must alter the value of the condition to allow the loop to finish.

```
/* Sample program including while  */
#include <stdio.h>

main()
{
        int  loop = 0;

        while( loop <= 10 ) {
                printf("%d\n", loop);
                ++loop;
        }
}
```

The above program uses a *while* loop to repeat the statements

```
printf("%d\n", loop);
++loop;
```

whilst the value of the variable *loop* is less than or equal to 10.

Note how the variable upon which the *while* is dependant is initialised prior to the *while* statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the *while* loop will terminate.

This program is functionally equivalent to the earlier *for* program which counted to ten.

THE DO WHILE STATEMENT
The *do { } while* statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero). The loop is executed as least once.

```
/* Demonstration of DO...WHILE    */
#include <stdio.h>

main()
{
        int  value, r_digit;

        printf("Enter the number to be reversed.\n");
        scanf("%d", &value);
        do {
                r_digit = value % 10;
                printf("%d", r_digit);
                value = value / 10;
        } while( value != 0 );
        printf("\n");
        }
```
The above program reverses a number that is entered by the user. It does this by using the modulus *%* operator to extract the right most digit into the variable *r_digit*. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

It is our contention that this programming construct is improper and should be avoided. It has potential problems, and you should be aware of these.

One such problem is deemed to be *lack of control*. Considering the above program code portion,

```
                do {
                        r_digit = value % 10;
                        printf("%d", r_digit);
                        value = value / 10;
                } while( value != 0 );
```
there is **NO** choice whether to execute the loop. Entry to the loop is automatic, as you only get a choice to continue.

Another problem is that the loop is always executed at least once. This is a by-product of the lack of control. This means its possible to enter a *do { } while* loop with invalid data.

Beginner programmers can easily get into a whole heap of trouble, so our advice is to avoid its use. This is the only time that you will encounter it in this course. Its easy to avoid the use of this construct by replacing it with the following algorithms,

```
        initialise loop control variable
        while( loop control variable is valid ) {
                process data
                adjust control variable if necessary
        }
```

Okay, lets now rewrite the above example to remove the *do { } while* construct.
```
        /* rewritten code to remove construct */
        #include <stdio.h>

        main()
        {
                int  value, r_digit;

                value = 0;
                while( value <= 0 ) {
                        printf("Enter the number to be reversed.\n");
                        scanf("%d", &value);
                        if( number <= 0 )
                                printf("The number must be positive\n");
                }

                while( value != 0 )
                {
                        r_digit = value % 10;
                        printf("%d", r_digit);
                        value = value / 10;
                }
                printf("\n");
        }
```

MAKING DECISIONS

SELECTION (IF STATEMENTS)

The *if* statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```
        if( expression )
                program statement;
```

Example;

```
        if( students < 65 )
                ++student_count;
```

In the above example, the variable *student_count* is incremented by one only if the value of the integer variable *students* is less than 65.

The following program uses an *if* statement to validate the users input to be in the range 1-10.

```
        #include <stdio.h>

        main()
        {
                int number;
                int valid = 0;

                while( valid == 0 ) {
                        printf("Enter a number between 1 and 10 -->");
                        scanf("%d", &number);
                        /* assume number is valid */
                        valid = 1;
                        if( number < 1 ) {
                                printf("Number is below 1. Please re-enter\n");
                                valid = 0;
                        }
                        if( number > 10 ) {
                                printf("Number is above 10. Please re-enter\n");
                                valid = 0;
                        }
                }
                printf("The number is %d\n", number );
        }
```

# EXERCISE C10:
Write a C program that allows the user to enter in 5 grades, ie, marks between 0 - 100. The program must calculate the average mark, and state the number of marks less than 65.

```
@       #include <stdio.h>

        main()
        {
                int grade;      /* to hold the entered grade */
                float average;  /* the average mark */
                int loop;       /* loop count */
                int sum;        /* running total of all entered grades */
                int valid_entry;        /* for validation of entered grade */
                int failures;   /* number of people with less than 65 */

                sum = 0;        /* initialise running total to 0 */
                failures = 0;

                for( loop = 0; loop < 5; loop = loop + 1 )
                {
                        valid_entry = 0;
                        while( valid_entry == 0 )
                        {
                                printf("Enter mark (1-100):");
                                scanf(" %d", &grade );
                                if ((grade > 1 ) || (grade < 100 ))
                                {
                                        valid_entry = 1;
                                }
                        }
                        if( grade < 65 )
                                failures++;
                        sum = sum + grade;
                }
                average = (float) sum / loop;
                printf("The average mark was %.2f\n", average );
                printf("The number less than 65 was %d\n", failures );
        }
```

Consider the following program which determines whether a character entered from the keyboard is within the range A to Z.

```
        #include <stdio.h>

        main()
        {
                char letter;

                printf("Enter a character -->");
                scanf(" %c", &letter );

                if( letter >= 'A' ) {
                        if( letter <= 'Z' )
                                printf("The character is within A to Z\n");
```

```
            }
    }
```

The program does not print any output if the character entered is not within the range A to Z. This can be addressed on the following pages with the *if else* construct.

Please note use of the leading space in the statement (before %c)

```
        scanf(" %c", &letter );
```

This enables the skipping of leading TABS, Spaces, (collectively called whitespaces) and the ENTER KEY. If the leading space was not used, then the first entered character would be used, and *scanf* would not ignore the whitespace characters.

COMPARING float types FOR EQUALITY

Because of the way in which float types are stored, it makes it very difficult to compare float types for equality. Avoid trying to compare float variables for equality, or you may encounter unpredictable results.

if else

The general format for these are,

```
         if( condition 1 )
            statement1;
         else if( condition 2 )
            statement2;
         else if( condition 3 )
            statement3;
         else
            statement4;
```

The *else* clause allows action to be taken where the condition evaluates as false (zero).

The following program uses an *if else* statement to validate the users input to be in the range 1-10.

```
        #include <stdio.h>

        main()
        {
                int number;
                int valid = 0;

                while( valid == 0 ) {
                        printf("Enter a number between 1 and 10 -->");
                        scanf("%d", &number);
                        if( number < 1 ) {
                                printf("Number is below 1. Please re-enter\n");
                                valid = 0;
                        }
                        else if( number > 10 ) {
                                printf("Number is above 10. Please re-enter\n");
                                valid = 0;
                        }
                        else
                                valid = 1;
                }
                printf("The number is %d\n", number );
        }
```

This program is slightly different from the previous example in that an *else* clause is used to set the variable *valid* to 1. In this program, the logic should be easier to follow.

```
        /* Illustates nested if else and multiple arguments to the scanf function.
*/
        #include <stdio.h>

        main()
        {
                int    invalid_operator = 0;
                char   operator;
                float  number1, number2, result;

                printf("Enter two numbers and an operator in the format\n");
                printf(" number1 operator number2\n");
                scanf("%f %c %f", &number1, &operator, &number2);

                if(operator == '*')
                        result = number1 * number2;
                else if(operator == '/')
                        result = number1 / number2;
                else if(operator == '+')
                        result = number1 + number2;
                else if(operator == '-')
                        result = number1 - number2;
                else
                        invalid_operator = 1;

                if( invalid_operator != 1 )
                        printf("%f %c %f is %f\n", number1, operator, number2,
result );
                else
                        printf("Invalid operator.\n");
        }
```

The above program acts as a simple calculator.

Practise Exercise 5: while loops and if else

1. Use a while loop to print the integer values 1 to 10 on the screen

```
12345678910
```

```
#include <stdio.h>

main()
{
        int loop;
        loop = 1;
        while( loop <= 10 ) {
                printf("%d", loop);
                loop++;
        }
        printf("\n");
}
```

2. Use a nested while loop to reproduce the following output

```
1
22
333
4444
55555
```

```
#include <stdio.h>

main()
{
        int loop;
        int count;
        loop = 1;
        while( loop <= 5 ) {
                count = 1;
                while( count <= loop ) {
                        printf("%d", count);
                        count++;
                }
                loop++;
        }
        printf("\n");
}
```

3. Use an if statement to compare the value of an integer called sum against the value 65, and if it is less, print the text string "Sorry, try again".

```
if( sum < 65 )
```

```
		printf("Sorry, try again.\n");
```

4. If total is equal to the variable good_guess, print the value of total, else print the value of good_guess.

```
	if( total == good_guess )
		printf("%d\n", total );
	else
		printf("%d\n", good_guess );
```

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

Combining more than one condition
These allow the testing of more than one condition as part of selection statements. The symbols are

```
	LOGICAL AND     &&
```
Logical and requires all conditions to evaluate as TRUE (non-zero).

```
	LOGICAL OR      ||
```
Logical or will be executed if any ONE of the conditions is TRUE (non-zero).

```
	LOGICAL NOT      !
```
logical not negates (changes from TRUE to FALSE, vsvs) a condition.

```
	LOGICAL EOR      ^
```
Logical eor will be excuted if either condition is TRUE, but NOT if they are all true.

The following program uses an *if* statement with logical AND to validate the users input to be in the range 1-10.

```
	#include <stdio.h>

	main()
	{
		int number;
		int valid = 0;

		while( valid == 0 ) {
			printf("Enter a number between 1 and 10 -->");
			scanf("%d", &number);
			if( (number < 1 ) || (number > 10) ){
				printf("Number is outside range 1-10. Please re-
enter\n");
				valid = 0;
			}
			else
				valid = 1;
		}
		printf("The number is %d\n", number );
	}
```
This program is slightly different from the previous example in that a LOGICAL AND eliminates one of the *else* clauses.

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

NEGATION

```
	#include <stdio.h>

	main()
	{
		int flag = 0;
		if( ! flag ) {
			printf("The flag is not set.\n");
			flag = ! flag;
		}
		printf("The value of flag is %d\n", flag);
	}
```

The program tests to see if *flag* is not (!) set; equal to zero. It then prints the appropriate message, changes the state of *flag*; *flag* becomes equal to not *flag*; equal to 1. Finally the value of *flag* is printed.

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

Range checking using Compound Relationals
Consider where a value is to be inputted from the user, and checked for validity to be within a certain range, lets say between the integer values 1 and 100.

```
	#include <stdio.h>

	main()
	{
		int number;
		int valid = 0;

		while( valid == 0 ) {
			printf("Enter a number between 1 and 100");
			scanf("%d", &number );
			if( (number < 1) || (number > 100) )
				printf("Number is outside legal range\n");
			else
				valid = 1;
		}
		printf("Number is %d\n", number );
	}
```

The program uses *valid*, as a flag to indicate whether the inputted data is within the required range of allowable values. The while loop continues whilst *valid* is 0.

The statement

```
if( (number < 1) || (number > 100) )
```

checks to see if the number entered by the user is within the valid range, and if so, will set the value of *valid* to 1, allowing the while loop to exit.

Now consider writing a program which validates a character to be within the range A-Z, in other words *alphabetic*.

```
#include <stdio.h>

main()
{
        char ch;
        int valid = 0;

        while( valid == 0 ) {
                printf("Enter a character A-Z");
                scanf(" %c", &ch );
                if( (ch >= 'A') || (ch <= 'Z') )
                        valid = 1;
                else
                        printf("Character is outside legal range\n");
        }
        printf("Character is %c\n", ch );
}
```

---

switch() case:
The *switch case* statement is a better way of writing a program when a series of *if elses* occurs. The general format for this is,

```
switch ( expression ) {
        case  value1:
                program statement;
                program statement;
                ......
                break;
        case  valuen:
                program statement;
                .......
                break;
        default:
                .......
                .......
                break;
}
```

The keyword *break* must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Rules for switch statements

```
values for 'case' must be integer or character constants
the order of the 'case' statements is unimportant
the default clause may occur first (convention places it last)
you cannot use expressions or ranges
```

```
#include <stdio.h>

main()
{
        int menu, numb1, numb2, total;

        printf("enter in two numbers -->");
        scanf("%d %d", &numb1, &numb2 );
        printf("enter in choice\n");
        printf("1=addition\n");
        printf("2=subtraction\n");
        scanf("%d", &menu );
        switch( menu ) {
                case 1: total = numb1 + numb2; break;
                case 2: total = numb1 - numb2; break;
                default: printf("Invalid option selected\n");
        }
        if( menu == 1 )
                printf("%d plus %d is %d\n", numb1, numb2, total );
        else if( menu == 2 )
                printf("%d minus %d is %d\n", numb1, numb2, total );
}
```

The above program uses a *switch* statement to validate and select upon the users input choice, simulating a simple menu of choices.

# EXERCISE C11:
Rewrite the previous program, which accepted two numbers and an operator, using the *switch case* statement.

@ Rewrite the previous program, which accepted two numbers and an operator, using the *switch case* statement.

```
        /* Illustates nested if else and multiple arguments to the scanf function.
*/
        #include <stdio.h>

        main()
        {
                int  invalid_operator = 0;
                char  operator;
                float  number1, number2, result;

                printf("Enter two numbers and an operator in the format\n");
                printf(" number1 operator number2\n");
                scanf("%f %c %f", &number1, &operator, &number2);

                if(operator == '*')
                        result = number1 * number2;
                else if(operator == '/')
                        result = number1 / number2;
                else if(operator == '+')
                        result = number1 + number2;
                else if(operator == '-')
                        result = number1 - number2;
                else
                        invalid_operator = 1;

                if( invalid_operator != 1 )
                        printf("%f %c %f is %f\n", number1, operator, number2,
result );
                else
                        printf("Invalid operator.\n");
        }
```

Solution

```
        /* Illustates switch */
        #include <stdio.h>

        main()
        {
                int  invalid_operator = 0;
                char  operator;
                float  number1, number2, result;

                printf("Enter two numbers and an operator in the format\n");
                printf(" number1 operator number2\n");
                scanf("%f %c %f", &number1, &operator, &number2);

                switch( operator ) {
                        case '*' : result = number1 * number2; break;
                        case '/' : result = number1 / number2; break;
                        case '+' : result = number1 + number2; break;
                        case '-' : result = number1 - number2; break;
                        default : invalid_operator = 1;
                }
                switch( invalid_operator ) {
                        case 1 : printf("Invalid operator.\n"); break;
```

```
                       default : printf("%f %c %f is %f\n", number1, operator,
number2, result );
               }
        }
```

Practise Exercise 6

Compound Relationals and switch

1. if sum is equal to 10 and total is less than 20, print the text string "incorrect.".

```
        if( (sum == 10) && (total < 20) )
                printf("incorrect.\n");
```

2. if flag is 1 or letter is not an 'X', then assign the value 0 to exit_flag, else set exit_flag to 1.

```
        if( (flag == 1) || (letter != 'X') )
                exit_flag = 0;
        else
                exit_flag = 1;
```

3. rewrite the following statements using a switch statement

```
        if( letter == 'X' )
                sum = 0;
        else if ( letter == 'Z' )
                valid_flag = 1;
        else if( letter == 'A' )
                sum = 1;
        else
                printf("Unknown letter -->%c\n", letter );


        switch( letter ) {
                case 'X' : sum = 0; break;
                case 'Z' : valid_flag = 1; break;
                case 'A' : sum = 1; break;
                default  : printf("Unknown letter -->%c\n", letter );
        }
```

ACCEPTING SINGLE CHARACTERS FROM THE KEYBOARD

getchar
The following program illustrates this,

```
#include <stdio.h>

main()
{
        int  i;
        int ch;

        for( i = 1; i<= 5; ++i ) {
                ch = getchar();
                putchar(ch);
        }
}
```

The program reads five characters (one for each iteration of the for loop) from the keyboard. Note that *getchar()* gets a single character from the keyboard, and *putchar()* writes a single character (in this case, *ch*) to the console screen.

The file ctype.h provides routines for manipulating characters.

BUILT IN FUNCTIONS FOR STRING HANDLING

string.h
You may want to look at the section on arrays first!. The following macros are built into the file string.h

strcat	*Appends a string*
strchr	*Finds first occurrence of a given character*
strcmp	*Compares two strings*
strcmpi	*Compares two strings, non-case sensitive*
strcpy	*Copies one string to another*
strlen	*Finds length of a string*
strlwr	*Converts a string to lowercase*
strncat	*Appends n characters of string*
strncmp	*Compares n characters of two strings*
strncpy	*Copies n characters of one string to another*
strnset	*Sets n characters of string to a given character*
strrchr	*Finds last occurrence of given character in string*
strrev	*Reverses string*
strset	*Sets all characters of string to a given character*
strspn	*Finds first substring from given character set in string*
strupr	*Converts string to uppercase*

To convert a string to uppercase

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

```
#include <string.h>

main()
{
        char name[80]; /* declare an array of characters 0-79 */

        printf("Enter in a name in lowercase\n");
        scanf( "%s", name );
        strupr( name );
        printf("The name is uppercase is %s", name );
}
```

BUILT IN FUNCTIONS FOR CHARACTER HANDLING
The following character handling functions are defined in ctype.h

isalnum	*Tests for alphanumeric character*
isalpha	*Tests for alphabetic character*
isascii	*Tests for ASCII character*
iscntrl	*Tests for control character*
isdigit	*Tests for 0 to 9*
isgraph	*Tests for printable character*
islower	*Tests for lowercase*
isprint	*Tests for printable character*
ispunct	*Tests for punctuation character*
isspace	*Tests for space character*
isupper	*Tests for uppercase character*
isxdigit	*Tests for hexadecimal*
toascii	*Converts character to ascii code*
tolower	*Converts character to lowercase*
toupper	*Converts character to uppercase*

To convert a string array to uppercase a character at a time using *toupper()*

```
#include <stdio.h>
#include <ctype.h>
main()
{
        char name[80];
        int loop;

        printf("Enter in a name in lowercase\n");
        scanf( "%s", name );
        for( loop = 0; name[loop] != 0; loop++ )
                name[loop] = toupper( name[loop] );

        printf("The name is uppercase is %s", name );
}
```

Validation Of User Input In C

Basic Rules

- Don't pass invalid data onwards.
- Validate data at input time.
- Always give the user meaningful feedback
- Tell the user what you expect to read as input

```
/* example one, a simple continue statement */
#include <stdio.h>
#include <ctype.h>

main()
{
        int     valid_input;    /* when 1, data is valid and loop is exited */
        char    user_input;     /* handles user input, single character menu 
choice */

        valid_input = 0;
        while( valid_input == 0 ) {
                printf("Continue (Y/N)?\n");
                scanf("  %c", &user_input );
                user_input = toupper( user_input );
                if((user_input == 'Y') || (user_input == 'N') )  valid_input = 1;
                else  printf("\007Error: Invalid choice\n");
        }
}
```

```
/* example two, getting and validating choices */
#include <stdio.h>
#include <ctype.h>

main()
{
        int     exit_flag = 0, valid_choice;
        char    menu_choice;

        while( exit_flag == 0 ) {
                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("\nC = Copy File\nE = Exit\nM = Move File\n");
                        printf("Enter choice:\n");
                        scanf("   %c", &menu_choice );
                        if((menu_choice=='C') || (menu_choice=='E') || 
(menu_choice=='M'))
                                valid_choice = 1;
                        else
                                printf("\007Error. Invalid menu choice 
selected.\n");
                }
                switch( menu_choice ) {
                        case 'C' : ....................();    break;
                        case 'E' : exit_flag = 1;  break;
                        case 'M' : ...................();  break;
                        default : printf("Error--- Should not occur.\n"); break;
                }
        }
}
```

<u>Other validation examples :</u>

Handling User Input In C
scanf() has problems, in that if a user is expected to type an integer, and types a string instead, often the program bombs. This can be overcome by reading all input as a string (use *getchar()*), and then converting the string to the correct data type.

```
/* example one, to read a word at a time */
#include <stdio.h>
#include <ctype.h>
#define MAXBUFFERSIZE   80

void cleartoendofline( void );  /* ANSI function prototype */

void cleartoendofline( void )
{
        char ch;
        ch = getchar();
        while( ch != '\n' )
                ch = getchar();
}

main()
{
        char    ch;                     /* handles user input */
        char    buffer[MAXBUFFERSIZE];  /* sufficient to handle one line */
        int     char_count;             /* number of characters read for this line
*/
        int     exit_flag = 0;
        int     valid_choice;

        while( exit_flag  == 0 ) {
                printf("Enter a line of text (<80 chars)\n");
                ch = getchar();
                char_count = 0;
                while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                        buffer[char_count++] = ch;
                        ch = getchar();
                }
                buffer[char_count] = 0x00;      /* null terminate buffer */
                printf("\nThe line you entered was:\n");
                printf("%s\n", buffer);

                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Continue (Y/N)?\n");
                        scanf(" %c", &ch );
                        ch = toupper( ch );
                        if((ch == 'Y') || (ch == 'N') )
                                valid_choice = 1;
                        else
                                printf("\007Error: Invalid choice\n");
                        cleartoendofline();
                }
                if( ch == 'N' ) exit_flag = 1;
        }
}
```

Another Example, read a number as a string

```
/* example two, reading a number as a string */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXBUFFERSIZE   80

void cleartoendofline( void );          /* ANSI function prototype */

void cleartoendofline( void )
{
        char ch;
        ch = getchar();
        while( ch != '\n' )
                ch = getchar();
}

main()
{
        char    ch;                     /* handles user input */
        char    buffer[MAXBUFFERSIZE];  /* sufficient to handle one line */
        int     char_count;             /* number of characters read for this line
*/
        int     exit_flag = 0, number, valid_choice;

        while( exit_flag  == 0 ) {
                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Enter a number between 1 and 1000\n");
                        ch = getchar();
                        char_count = 0;
                        while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                                buffer[char_count++] = ch;
                                ch = getchar();
                        }
                        buffer[char_count] = 0x00;      /* null terminate buffer */
                        number = atoi( buffer );
                        if( (number < 1) || (number > 1000) )
                                printf("\007Error. Number outside range 1-1000\n");
                        else
                                valid_choice = 1;
                }
                printf("\nThe number you entered was:\n");
                printf("%d\n", number);
                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Continue (Y/N)?\n");
                        scanf(" %c", &ch );
                        ch = toupper( ch );
                        if((ch == 'Y') || (ch == 'N') )
                                valid_choice = 1;
                        else
                                printf("\007Error: Invalid choice\n");
                        cleartoendofline();
                }
                if( ch == 'N' ) exit_flag = 1;
        }
}
```

THE CONDITIONAL EXPRESSION OPERATOR
This conditional expression operator takes THREE operators. The two symbols used to denote this operator are the ? and the :. The first operand is placed before the ?, the second operand between the ? and the :, and the third after the :. The general format is,

```
        condition ? expression1 : expression2
```

If the result of condition is TRUE (non-zero), expression1 is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then expression2 is evaluated and its result becomes the result of the operation. An example will help,

```
        s = ( x < 0 ) ? -1 : x * x;

        If x is less than zero then s = -1
        If x is greater than zero then s = x * x
```

Example program illustrating conditional expression operator

```
        #include <stdio.h>

        main()
        {
                int input;

                printf("I will tell you if the number is positive, negative or 
zero!"\n");
                printf("please enter your number now--->");
                scanf("%d", &input );
```

```
              (input < 0) ? printf("negative\n") : ((input > 0) ?
printf("positive\n") : printf("zero\n"));
        }
```

# EXERCISE C12:
Evaluate the following expression, where a=4, b=5

```
        least_value = ( a < b ) ? a : b;
```

@ Evaluate the following expression, where a=4, b=5

```
        least_value = ( a < b ) ? a : b;
```

max_value = 5

ARRAYS

Little Boxes on the hillside
Arrays are a data structure which hold multiple variables of the same data type. Consider the case where a programmer needs to keep track of a number of people within an organisation. So far, our initial attempt will be to create a specific variable for each user. This might look like,

```
        int name1 = 101;
        int name2 = 232;
        int name3 = 231;
```

It becomes increasingly more difficult to keep track of this as the number of variables increase. Arrays offer a solution to this problem.

An array is a multi-element box, a bit like a filing cabinet, and uses an indexing system to find each variable stored within it. In C, indexing starts at **zero**.

Arrays, like other variables in C, must be declared before they can be used.

The replacement of the above example using arrays looks like,

```
        int names[4];
        names[0] = 101;
        names[1] = 232;
        names[2] = 231;
        names[3] = 0;
```

We created an array called *names*, which has space for four integer variables. You may also see that we stored 0 in the last space of the array. This is a common technique used by C programmers to signify the end of an array.

Arrays have the following syntax, using square brackets to access each indexed value (called an **element**).

```
x[i]
```

so that *x[5]* refers to the sixth element in an array called *x*. In C, array elements start with 0. Assigning values to array elements is done by,

```
x[10] = g;
```

and assigning array elements to a variable is done by,

```
g = x[10];
```

In the following example, a character based array named *word* is declared, and each element is assigned a character. The last element is filled with a zero value, to signify the end of the character string (in C, there is no string type, so character based arrays are used to hold strings). A printf statement is then used to print out all elements of the array.

```
/*  Introducing array's, 2  */
#include <stdio.h>

main()
{
        char word[20];

        word[0] = 'H';
        word[1] = 'e';
        word[2] = 'l';
        word[3] = 'l';
        word[4] = 'o';
        word[5] = 0;
        printf("The contents of word[] is -->%s\n", word );
}
```

DECLARING ARRAYS
Arrays may consist of any of the valid data types. Arrays are declared along with all other variables in the declaration section of the program.

```
/*  Introducing array's  */
#include <stdio.h>
```

```
main()
{
        int   numbers[100];
        float  averages[20];

        numbers[2] = 10;
        --numbers[2];
        printf("The 3rd element of array numbers is %d\n", numbers[2]);
}
```

The above program declares two arrays, assigns 10 to the value of the 3rd element of array *numbers*, decrements this value (*--numbers[2]*), and finally prints the value. The number of elements that each array is to have is included inside the square brackets.

ASSIGNING INITIAL VALUES TO ARRAYS

The declaration is preceded by the word *static*. The initial values are enclosed in braces, eg,

```
#include <stdio.h>
main()
{
        int x;
        static int  values[] = { 1,2,3,4,5,6,7,8,9 };
        static char  word[] = { 'H','e','l','l','o' };
        for( x = 0; x < 9; ++x )
                printf("Values [%d] is %d\n", x, values[x]);
}
```

The previous program declares two arrays, *values* and *word*. Note that inside the squarebrackets there is no variable to indicate how big the array is to be. In this case, C initializes the array to the number of elements that appear within the initialize braces. So values consist of 9 elements (numbered 0 to 8) and the char array *word* has 5 elements.

The following program shows how to initialise all the elements of an integer based array to the value 10, using a for loop to cycle through each element in turn.

```
#include <stdio.h>
main()
{
        int count;
        int  values[100];
        for( count = 0; count < 100; count++ )
                values[count] = 10;
}
```

MULTI DIMENSIONED ARRAYS
Multi-dimensioned arrays have two or more index values which specify the element in the array.

```
multi[i][j]
```

In the above example, the first index value *i* specifies a row index, whilst *j* specifies a column index.

<H5**Declaration and calculations**

```
int        m1[10][10];
static int m2[2][2] = { {0,1}, {2,3} };

sum = m1[i][j] + m2[k][l];
```

NOTE the strange way that the initial values have been assigned to the two-dimensional array m2. Inside the braces are,

```
{ 0, 1 },
{ 2, 3 }
```

Remember that arrays are split up into row and columns. The first is the row, the second is the column. Looking at the initial values assigned to *m2*, they are,

```
m2[0][0] = 0
m2[0][1] = 1
m2[1][0] = 2
m2[1][1] = 3
```

# EXERCISE C13:
Given a two dimensional array, write a program that totals all elements, printing the total.

@ Given a two dimensional array write a program that totals all elements printing the total.

```
#include <stdio.h>

main()
{
        static int m[][] = { {10,5,-3}, {9, 0, 0}, {32,20,1}, {0,0,8} };
        int row, column, sum;

        sum = 0;
        for( row = 0; row < 4; row++ )
```

```
                for( column = 0; column < 3; column++ )
                        sum = sum + m[row][column];
        printf("The total is %d\n", sum );
}
```

# EXERCISE C14:
What value is assigned to the elements which are not assigned initialised.

@ They get initialised to **ZERO**

CHARACTER ARRAYS [STRINGS]
Consider the following program,

```
#include <stdio.h>
main()
{
        static char name1[] = {'H','e','l','l','o'};
        static char name2[] = "Hello";
        printf("%s\n", name1);
        printf("%s\n", name2);
}
```

The difference between the two arrays is that *name2* has a null placed at the end of the string, ie, in name2[5], whilst *name1* has not. To insert a null at the end of the name1 array, the initialization can be changed to,

```
static char name1[] = {'H','e','l','l','o','\0'};
```

Consider the following program, which initialises the contents of the character based array *word* during the program, using the function *strcpy*, which necessitates using the include file *string.h*

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

main()
{
        char word[20];

        strcpy( word, "hi there." );
        printf("%s\n", word );
}
```

SOME VARIATIONS IN DECLARING ARRAYS

```
        int  numbers[10];

        static int numbers[10] = { 34, 27, 16 };

        static int numbers[] = { 2, -3, 45, 79, -14, 5, 9, 28, -1, 0 };

        static char text[] = "Welcome to New Zealand.";

        static float radix[12] = { 134.362, 1913.248 };

        double  radians[1000];
```

READING CHARACTER STRINGS FROM THE KEYBOARD
Character based arrays are often referred to in C as strings. C does not support a string type, so *scanf()* is used to handle character based arrays. This assumes that a 0 or NULL value is stored in the last element of the array. Consider the following, which reads a string of characters (excluding spaces) from the keyboard.

```
        char string[18];
        scanf("%s", string);
```

NOTE that the & character does not need to precede the variable name when the formatter %s is used! If the users response was

```
        Hello
```

then

```
        string[0] = 'H'
        string[1] = 'e'
        ....
        string[4] = 'o'
        string[5] = '\0'
```

Practise Exercise 7: Arrays

1. Declare a character based array called letters of ten elements

```
        char letters[10];
```

2. Assign the character value 'Z' to the fourth element of the letters array

```
        letters[3] = 'Z';
```

3. Use a for loop to total the contents of an integer array called numbers which has five elements. Store the result in an integer called total.

```
        for( loop = 0, total = 0; loop < 5; loop++ )
                total = total + numbers[loop];
```

4. Declare a multidimensioned array of floats called balances having three rows and five columns.

```
        float balances[3][5];
```

5. Write a for loop to total the contents of the multidimensioned float array balances.

```
        for( row = 0, total = 0; row < 3; row++ )
                for( column = 0; column < 5; column++ )
                        total = total + balances[row][column];
```

6. Assign the text string "Hello" to the character based array words at declaration time.

```
        static char words[] = "Hello";
```

7. Assign the text string "Welcome" to the character based array stuff (not at declaration time)

```
        char stuff[50];

        strcpy( stuff, "Welcome" );
```

8. Use a printf statement to print out the third element of an integer array called totals

```
        printf("%d\n", totals[2] );
```

9. Use a printf statement to print out the contents of the character array called words

```
	printf("%s\n", words);
```

10. Use a scanf statement to read a string of characters into the array words.

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

11. Write a for loop which will read five characters (use scanf) and deposit them into the character based array words, beginning at element 0.

```
	for( loop = 0; loop < 5; loop++ )
		scanf("%c", &words[loop] );
```

FUNCTIONS
A function in C can perform a particular task, and supports the concept of modular programming design techniques.

We have already been exposed to functions. The main body of a C program, identified by the keyword *main*, and enclosed by the left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

Functions have a basic structure. Their format is

```
	return_data_type  function_name  ( arguments, arguments )
	data_type_declarations_of_arguments;
	{
		function_body
	}
```

It is worth noting that a return_data_type is assumed to be type *int* unless otherwise specified, thus the programs we have seen so far imply that *main()* returns an integer to the operating system.

ANSI C varies slightly in the way that functions are declared. Its format is

```
	return_data_type function_name (data_type variable_name, data_type 
variable_name, .. )
	{
		function_body
	}
```

This permits type checking by utilizing function prototypes to inform the compiler of the type and number of parameters a function accepts. When calling a function, this information is used to perform type and parameter checking.

ANSI C also requires that the return_data_type for a function which does not return data must be type *void*. The default return_data_type is assumed to be integer unless otherwise specified, but must match that which the function declaration specifies.

A simple function is,

```
void print_message( void )
{
        printf("This is a module called print_message.\n");
}
```

Note the function name is *print_message*. No arguments are accepted by the function, this is indicated by the keyword *void* in the accepted parameter section of the function declaration. The return_data_type is void, thus data is not returned by the function.

An ANSI C function prototype for *print_message()* is,

```
void print_message( void );
```

Function prototypes are listed at the beginning of the source file. Often, they might be placed in a users .h (header) file.

FUNCTIONS
Now lets incorporate this function into a program.

```
/* Program illustrating a simple function call */
#include <stdio.h>

void print_message( void );    /* ANSI C function prototype */

void print_message( void )     /* the function code */
{
        printf("This is a module called print_message.\n");
}

main()
{
        print_message();
}
```

To call a function, it is only necessary to write its name. The code associated with the function name is executed at that point in the program. When the function terminates, execution begins with the statement which follows the function name.

In the above program, execution begins at *main()*. The only statement inside the main body of the program is a call to the code of function *print_message()*. This code is executed, and when finished returns back to *main()*.

As there is no further statements inside the main body, the program terminates by returning to the operating system.

FUNCTIONS
In the following example, the function accepts a single data variable, but does not return any information.

```
/* Program to calculate a specific factorial number  */
#include <stdio.h>

void calc_factorial( int );    /* ANSI function prototype */

void calc_factorial( int n )
{
        int  i, factorial_number = 0;

        for( i = 1; i <= n; ++i )
                factorial_number *= i;

        printf("The factorial of %d is %d\n", n, factorial_number );
}

main()
{
        int  number = 0;

        printf("Enter a number\n");
        scanf("%d", &number );
        calc_factorial( number );
}
```

Lets look at the function *calc_factorial()*. The declaration of the function

```
void calc_factorial( int n )
```

indicates there is no return data type and a single integer is accepted, known inside the body of the function as *n*. Next comes the declaration of the local variables,

```
        int  i, factorial_number = 0;
```

It is more correct in C to use,

```
        auto int  i, factorial_number = 0;
```

as the keyword *auto* designates to the compiler that the variables are local. The program works by accepting a variable from the keyboard which is then passed to the function. In other words, the variable *number* inside the main body is then copied to the variable *n* in the function, which then calculates the correct answer.

RETURNING FUNCTION RESULTS
This is done by the use of the keyword *return*, followed by a data variable or constant value, the data type of which must match that of the declared return_data_type for the function.

```
        float add_numbers( float n1, float n2 )
        {
                return n1 + n2;     /* legal */
                return 6;           /* illegal, not the same data type */
                return 6.0;         /* legal */
        }
```

It is possible for a function to have multiple return statements.

```
        int validate_input( char command )
        {
                switch( command ) {
                        case '+' :
                        case '-' : return 1;
                        case '*' :
                        case '/' : return 2;
                        default  : return 0;
                }
        }
```

Here is another example

```
        /* Simple multiply program using argument passing */
        #include <stdio.h>

        int calc_result( int, int );            /* ANSI function prototype */

        int calc_result( int numb1, int numb2 )
        {
                auto int result;
                result = numb1 * numb2;
                return result;
        }

        main()
        {
                int  digit1 = 10, digit2 = 30, answer = 0;
                answer = calc_result( digit1, digit2 );
                printf("%d multiplied by %d is %d\n", digit1, digit2, answer );
        }
```

NOTE that the value which is returned from the function (ie result) must be declared in the function.

NOTE: The formal declaration of the function name is preceded by the data type which is returned,

```
int  calc_result ( numb1, numb2 )
```

# EXERCISE C15:
Write a program in C which incorporates a function using parameter passing and performs the addition of three numbers. The main section of the program is to print the result.

@ Write a program in C which incorporates a function using parameter passing and performs the addition of three numbers. The main section of the program is to print the result.

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

int calc_result( int var1, int var2, int var3 )
{
  int sum;

  sum = var1 + var2 + var3;
  return( sum );           /* return( var1 + var2 + var3 ); */
}

main()
{
  int numb1 = 2, numb2 = 3, numb3=4, answer=0;

  answer = calc_result( numb1, numb2, numb3 );
  printf("%d + %d + %d = %d\n", numb1, numb2, numb3, answer);
}
```

RETURNING FUNCTION RESULTS

This is done by the use of the keyword *return*, followed by a data variable or constant value, the data type of which must match that of the declared return_data_type for the function.

```
float add_numbers( float n1, float n2 )
{
        return n1 + n2;     /* legal */
        return 6;           /* illegal, not the same data type */
        return 6.0;         /* legal */
}
```

It is possible for a function to have multiple return statements.

```
int validate_input( char command )
{
        switch( command ) {
                case '+' :
                case '-' : return 1;
                case '*' :
                case '/' : return 2;
                default  : return 0;
        }
}
```

Here is another example

```
/* Simple multiply program using argument passing */
#include <stdio.h>

int calc_result( int, int );            /* ANSI function prototype */

int calc_result( int numb1, int numb2 )
{
        auto int result;
        result = numb1 * numb2;
        return result;
}

main()
{
        int  digit1 = 10, digit2 = 30, answer = 0;
        answer = calc_result( digit1, digit2 );
        printf("%d multiplied by %d is %d\n", digit1, digit2, answer );
}
```

NOTE that the value which is returned from the function (ie result) must be declared in the function.

NOTE: The formal declaration of the function name is preceded by the data type which is returned,

```
        int  calc_result ( numb1, numb2 )
```

# EXERCISE C15:
Write a program in C which incorporates a function using parameter passing and performs the addition of three numbers. The main section of the program is to print the result.

```
@       #include <stdio.h>
        int calc_result( int, int, int );

        int calc_result( int var1, int var2, int var3 )
        {
          int sum;

          sum = var1 + var2 + var3;
          return( sum );          /* return( var1 + var2 + var3 ); */
        }

        main()
        {
          int numb1 = 2, numb2 = 3, numb3=4, answer=0;

          answer = calc_result( numb1, numb2, numb3 );
          printf("%d + %d + %d = %d\n", numb1, numb2, numb3, answer);
        }
```

LOCAL AND GLOBAL VARIABLES

Local
These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global
These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

DEFINING GLOBAL VARIABLES

```
        /* Demonstrating Global variables  */
        #include <stdio.h>
        int add_numbers( void );                /* ANSI function prototype */

        /* These are global variables and can be accessed by functions from this
point on */
        int  value1, value2, value3;
        int add_numbers( void )
        {
                auto int result;
                result = value1 + value2 + value3;
                return result;
        }

        main()
        {
                auto int result;

                result = add_numbers();
                printf("The sum of %d + %d + %d is %d\n",
                        value1, value2, value3, final_result);
        }
```

The scope of global variables can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

```
#include <stdio.h>
void no_access( void ); /* ANSI function prototype */
void all_access( void );

static int n2;         /* n2 is known from this point onwards */

void no_access( void )
{
        n1 = 10;        /* illegal, n1 not yet known */
        n2 = 5;         /* valid */
}

static int n1;         /* n1 is known from this point onwards */

void all_access( void )
{
        n1 = 10;        /* valid */
        n2 = 3;         /* valid */
}
```

AUTOMATIC AND STATIC VARIABLES
C programs have a number of segments (or areas) where data is located. These segments are typically,

```
_DATA   Static data
_BSS    Uninitialized static data, zeroed out before call to main()
_STACK  Automatic data, resides on stack frame, thus local to functions
_CONST  Constant data, using the ANSI C keyword const
```

The use of the appropriate keyword allows correct placement of the variable onto the desired data segment.

```
        /* example program illustrates difference between static and automatic
variables */
        #include <stdio.h>
        void demo( void );              /* ANSI function prototypes */

        void demo( void )
        {
                auto int avar = 0;
                static int svar = 0;

                printf("auto = %d, static = %d\n", avar, svar);
                ++avar;
                ++svar;
        }


        main()
        {
                int i;

                while( i < 3 ) {
                        demo();
                        i++;
                }
        }
```

AUTOMATIC AND STATIC VARIABLES

```
        /* example program illustrates difference between static and automatic
variables */
        #include <stdio.h>
        void demo( void );              /* ANSI function prototypes */

        void demo( void )
        {
                auto int avar = 0;
                static int svar = 0;

                printf("auto = %d, static = %d\n", avar, svar);
                ++avar;
                ++svar;
        }


        main()
        {
                int i;

                while( i < 3 ) {
                        demo();
                        i++;
                }
        }
```

Program output

```
        auto = 0, static = 0
        auto = 0, static = 1
        auto = 0, static = 2
```

Static variables are created and initialized once, on the first call to the function. Subsequent calls to the function do not recreate or re-initialize the static variable. When the function terminates, the variable still exists on the _DATA segment, but cannot be accessed by outside functions.

Automatic variables are the opposite. They are created and re-initialized on each entry to the function. They disappear (are de-allocated) when the function terminates. They are created on the _STACK segment.

PASSING ARRAYS TO FUNCTIONS
The following program demonstrates how to pass an array to a function.

```
        /* example program to demonstrate the passing of an array */
        #include <stdio.h>
        int maximum( int [] );          /* ANSI function prototype */

        int  maximum( int values[5] )
        {
                int  max_value, i;

                max_value = values[0];
                for( i = 0; i < 5; ++i )
                        if( values[i] > max_value )
                                max_value = values[i];

                return max_value;
        }

        main()
        {
                int values[5], i, max;

                printf("Enter 5 numbers\n");
                for( i = 0; i < 5; ++i )
                        scanf("%d", &values[i] );

                max = maximum( values );
                printf("\nMaximum value is %d\n", max );
        }
```

Note: The program defines an array of five elements (values) and initializes each element to the users inputted values. The array *values* is then passed to the function. The declaration

```
	int  maximum( int values[5] )
```

defines the function name as *maximum*, and declares that an integer is passed back as the result, and that it accepts a data type called *values*, which is declared as an array of five integers. The values array in the main body is now known as the array values inside function maximum. **IT IS NOT A COPY, BUT THE ORIGINAL**.

This means any changes will update the original array.

A local variable *max_value* is set to the first element of values, and a for loop is executed which cycles through each element in values and assigns the lowest item to *max_value*. This number is then passed back by the return statement, and assigned to *max* in the main section.

Functions and Arrays

C allows the user to build up a library of modules such as the maximum value found in the previous example.

However, in its present form this module or function is limited as it only accepts ten elements. It is thus desirable to modify the function so that it also accepts the number of elements as an argument also. A modified version follows,

```
	/* example program to demonstrate the passing of an array */
	#include <stdio.h>

	int findmaximum( int [], int );            /* ANSI function prototype */

	int  findmaximum( int numbers[], int elements )
	{
		int  largest_value, i;

		largest_value = numbers[0];

		for( i = 0; i < elements; ++i )
			if( numbers[i] < largest_value )
				largest_value = numbers[i];

		return largest_value;
	}

	main()
	{
		static int numb1[] = { 5, 34, 56, -12, 3, 19 };
		static int numb2[] = { 1, -2, 34, 207, 93, -12 };

		printf("maximum of numb1[] is %d\n", findmaximum(numb1, 6));
		printf("maximum is numb2[] is %d\n", findmaximum(numb2, 6));
```

```
        }
```

PASSING OF ARRAYS TO FUNCTIONS
If an entire array is passed to a function, any changes made also occur to the original array.

PASSING OF MULTIDIMENSIONAL ARRAYS TO FUNCTIONS
If passing a multidimensional array, the number of columns must be specified in the formal parameter declaration section of the function.

# EXERCISE C16:

Write a C program incorporating a function to add all elements of a two dimensional array. The number of rows are to be passed to the function, and it passes back the total sum of all elements (Use at least a 4 x 4 array).

```
@       #include <stdio.h>

        int add2darray( int [][5], int );     /* function prototype */

        int add2darray( int array[][5], int rows )
        {
                int total = 0, columns, row;

                for( row = 0; row < rows; row++ )
                        for( columns = 0; columns < 5; columns++ )
                                total = total + array[row][columns];
                return total;
        }

        main()
        {
                int numbers[][] = { {1, 2, 35, 7, 10}, {6, 7, 4, 1, 0} };
                int sum;

                sum = add2darray( numbers, 2 );
                printf("the sum of numbers is %d\n", sum );
        }
```

FUNCTION PROTOTYPES
These have been introduced into the C language as a means of provided type checking and parameter checking for function calls. Because C programs are generally split up over a number of different source files which are independently compiled, then linked together to generate a run-time program, it is possible for errors to occur.

Consider the following example.

```
/* source file add.c */
void add_up( int numbers[20] )
{
        ....
}

/* source file mainline.c */
static float values[] = { 10.2, 32.1, 0.006, 31.08 };

main()
{
        float result;
        ...
        result = add_up( values );
}
```

As the two source files are compiled separately, the compiler generates correct code based upon what the programmer has written. When compiling mainline.c, the compiler assumes that the function add_up accepts an array of float variables and returns a float. When the two portions are combined and ran as a unit, the program will definitely not work as intended.

To provide a means of combating these conflicts, ANSI C has function prototyping. Just as data types need to be declared, functions are declared also. The function prototype for the above is,

```
/* source file mainline.c */
void add_up( int numbers[20] );
```

NOTE that the function prototype ends with a semi-colon; in this way we can tell its a declaration of a function type, not the function code. If mainline.c was re-compiled, errors would be generated by the call in the main section which references *add_up()*.

Generally, when developing a large program, a separate file would be used to contain all the function prototypes. This file can then be included by the compiler to enforce type and parameter checking.

ADDITIONAL ASSIGNMENT OPERATOR
Consider the following statement,

```
numbers[loop] += 7;
```

This assignment += is equivalent to multiply equals. It takes the value of *numbers[loop]*, adds it by 7, then assigns the value to *numbers[loop]*. In other words it is the same as,

```
numbers[loop] = numbers[loop] + 7;
```

# EXERCISE C17:
What is the outcome of the following, assuming time=2, a=3, b=4, c=5

```
        time -= 5;
        a *= b + c;
```

@

```
        time -= 5;
        a *= b + c;
```

```
        time = -3
        a = 27
```

SIMPLE EXCHANGE SORT ALGORITHM
The following steps define an algorithm for sorting an array,

```
        1. Set i to 0
        2. Set j to i + 1
        3. If a[i] > a[j], exchange their values
        4. Set j to j + 1. If j < n goto step 3
        5. Set i to i + 1. If i < n - 1 goto step 2
        6. a is now sorted in ascending order.

        Note: n is the number of elements in the array.
```

# EXERCISE C18:

Implement the above algorithm as a function in C, accepting the array and its size, returning the sorted array in ascending order so it can be printed out by the calling module. The array should consist of ten elements**.**

@ **A SIMPLE EXCHANGE SORT ALGORITHM**

The following steps define an algorithm for sorting an array,

```
        1. Set i to 0
        2. Set j to i + 1
        3. If a[i] > a[j], exchange their values
        4. Set j to j + 1. If j < n goto step 3
        5. Set i to i + 1. If i < n - 1 goto step 2
        6. a is now sorted in ascending order.

        Note: n is the number of elements in the array.
```

EXERCISE C18
Implement the above algorithm as a function in C, accepting the array and its size, returning the sorted array in ascending order so it can be printed out by the calling module. The array should consist of ten elements.

```
#include <stdio.h>

void sort( [], int );

void sort( array[], int elements )
{
        int i, j, temp;

        i = 0;
        while( i < (elements - 1) ) {
                j = i + 1;
                while( j < elements ) {
                        if( a[i] > a[j] ) {
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                        j++;
                }
                i++;
        }
}

main()
{
        int numbers[] = { 10, 9, 8, 23, 19, 11, 2, 7, 1, 13, 12 };
        int loop;

        printf("Before the sort the array was \n");
        for( loop = 0; loop < 11; loop++ )
                printf(" %d ", numbers[loop] );
        sort( numbers, 11 );
        printf("After the sort the array was \n");
        for( loop = 0; loop < 11; loop++ )
                printf(" %d ", numbers[loop] );
}
```

RECURSION
This is where a function repeatedly calls itself to perform calculations. Typical applications are games and Sorting trees and lists.

Consider the calculation of 6! (6 factorial)

```
 ie 6! = 6 * 5 * 4 * 3 * 2 * 1
    6! = 6 * 5!
    6! = 6 * ( 6 - 1 )!
    n! = n * ( n - 1 )!

        /* bad example for demonstrating recursion */
        #include <stdio.h>

        long int factorial( long int );         /* ANSI function prototype */
```

```
long int  factorial( long int n )
{
        long int result;

        if( n == 0L )
                result = 1L;
        else
                result = n * factorial( n - 1L );
        return ( result );
}

main()
{
        int j;

        for( j = 0; j < 11; ++j )
                printf("%2d! = %ld\n", factorial( (long) j) );
}
```

RECURSIVE PROGRAMMING: EXERCISE C19
Rewrite example c9 using a recursive function.

```
#include <stdio.h>
long int triang_rec( long int );

long int triang_rec( long int number )
{
    long int result;

    if( number == 0l )
      result = 0l;
    else
      result = number + triang_rec( number - 1 );
    return( result );
}

main ()
{
  int request;
  long int triang_rec(), answer;

  printf("Enter number to be calculated.\n");
  scanf( "%d", &request);

  answer = triang_rec( (long int) request );
  printf("The triangular answer is %l\n", answer);
}
```

```
Note this version of function triang_rec

#include <stdio.h>
long int triang_rec( long int );

long int triang_rec( long int number )
{
   return((number == 0l) ? 0l : number*triang_rec( number-1));
```

Practise Exercise 8: Functions

1. Write a function called menu which prints the text string "Menu choices". The function does not pass any data back, and does not accept any data as parameters.

```
void menu( void )
{
       printf("Menu choices");
}
```

2. Write a function prototype for the above function.

```
void menu( void );
```

3. Write a function called print which prints a text string passed to it as a parameter (ie, a character based array).

```
void print( char message[] )
{
       printf("%s, message );
}
```

4. Write a function prototype for the above function print.

```
void print( char [] );
```

5. Write a function called total, which totals the sum of an integer array passed to it (as the first parameter) and returns the total of all the elements as an integer. Let the second parameter to the function be an integer which contains the number of elements of the array.

```
int total( int array[], int elements )
{
        int loop, sum;

        for( loop = 0, sum = 0; loop < elements; loop++ )
                sum += array[loop];
        return sum;
}
```

6. Write a function prototype for the above function.

```
int total( int [], int );
```

Handling User Input In C
scanf() has problems, in that if a user is expected to type an integer, and types a string instead, often the program bombs. This can be overcome by reading all input as a string (use *getchar()*), and then converting the string to the correct data type.

```
/* example one, to read a word at a time */
#include <stdio.h>
#include <ctype.h>
#define MAXBUFFERSIZE   80

void cleartoendofline( void );  /* ANSI function prototype */

void cleartoendofline( void )
{
        char ch;
        ch = getchar();
        while( ch != '\n' )
                ch = getchar();
}

main()
{
        char    ch;                     /* handles user input */
        char    buffer[MAXBUFFERSIZE];  /* sufficient to handle one line */
        int     char_count;             /* number of characters read for this line
*/
        int     exit_flag = 0;
        int     valid_choice;

        while( exit_flag  == 0 ) {
                printf("Enter a line of text (<80 chars)\n");
                ch = getchar();
                char_count = 0;
                while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                        buffer[char_count++] = ch;
                        ch = getchar();
                }
                buffer[char_count] = 0x00;      /* null terminate buffer */
                printf("\nThe line you entered was:\n");
                printf("%s\n", buffer);

                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Continue (Y/N)?\n");
                        scanf(" %c", &ch );
                        ch = toupper( ch );
                        if((ch == 'Y') || (ch == 'N') )
                                valid_choice = 1;
                        else
                                printf("\007Error: Invalid choice\n");
                        cleartoendofline();
                }
                if( ch == 'N' ) exit_flag = 1;
        }
}
```

Another Example, read a number as a string

```
/* example two, reading a number as a string */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXBUFFERSIZE   80

void cleartoendofline( void );          /* ANSI function prototype */

void cleartoendofline( void )
{
        char ch;
        ch = getchar();
        while( ch != '\n' )
                ch = getchar();
}

main()
{
        char    ch;                     /* handles user input */
        char    buffer[MAXBUFFERSIZE];  /* sufficient to handle one line */
        int     char_count;             /* number of characters read for this line
*/
        int     exit_flag = 0, number, valid_choice;

        while( exit_flag  == 0 ) {
                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Enter a number between 1 and 1000\n");
                        ch = getchar();
                        char_count = 0;
                        while( (ch != '\n')  &&  (char_count < MAXBUFFERSIZE)) {
                                buffer[char_count++] = ch;
                                ch = getchar();
                        }
                        buffer[char_count] = 0x00;      /* null terminate buffer */
                        number = atoi( buffer );
                        if( (number < 1) || (number > 1000) )
                                printf("\007Error. Number outside range 1-1000\n");
                        else
                                valid_choice = 1;
                }
                printf("\nThe number you entered was:\n");
                printf("%d\n", number);

                valid_choice = 0;
                while( valid_choice == 0 ) {
                        printf("Continue (Y/N)?\n");
                        scanf(" %c", &ch );
                        ch = toupper( ch );
                        if((ch == 'Y') || (ch == 'N') )
                                valid_choice = 1;
                        else
                                printf("\007Error: Invalid choice\n");
                        cleartoendofline();
                }
                if( ch == 'N' ) exit_flag = 1;
        }
}
```

More Data Validation
Consider the following program

```
    #include <stdio.h>

    main() {
            int number;

            printf("Please enter a number\n");
            scanf("%d", &number );
            printf("The number you entered was %d\n", number );
    }
```

The above program has several problems

- the input is not validated to see if its the correct data type
- it is not clear if there are explicit number ranges expected
- the program might crash if an incorrect data type was entered

Perhaps the best way of handling input in C programs is to treat all input as a sequence of characters, and then perform the necessary data conversion.

At this point we shall want to explore some other aspects also, like the concepts of

- trapping data at the source
- the domino/ripple effect

Trapping Data At The Source
This means that the validation of data as to its correct range/limit and data type is best done at the point of entry. The benefits of doing this at the time of data entry are

- less cost later in the program maintenance phase (because data is already validated)
- programs are easier to maintain and modify
- reduces the chances of incorrect data crashing the program later on

The Ripple Through Effect
This refers to the problem of incorrect data which is allowed to propagate through the program. An example of this is sending invalid data to a function to process.

By trapping data at the source, and ensuring that it is correct as to its data type and range, we ensure that bad data cannot be passed onwards. This makes the code which works on processing the data simpler to write and thus reduces errors.

An example
Lets look at the case of wanting to handle user input. Now, we know that users of programs out there in user-land are a bunch of annoying people who spend most of their time inventing new and more wonderful ways of making our programs crash.

Lets try to implement a sort of general purpose way of handling data input, as a replacement to *scanf()*. To do this, we will implement a function which reads the input as a sequence of characters.

The function is *readinput()*, which, in order to make it more versatile, accepts several parameters,

- a character array to store the inputted data
- an integer which specifies the data type to read, STRING, INTEGER, ALPHA
- an integer which specifies the amount of digits/characters to read

We have used the some of the functions covered in ctype.h to check the data type of the inputted data.

```
/* version 1.0 */
#include <stdio.h>
#include <ctype.h>

#define MAX      80          /* maximum length of buffer           */
#define DIGIT     1          /* data will be read as digits 0-9    */
#define ALPHA     2          /* data will be read as alphabet A-Z */
#define STRING    3          /* data is read as ASCII              */

void readinput( char buff[], int mode, int limit ) {
        int ch, index = 0;

        ch = getchar();
        while( (ch != '\n') && (index < limit) ) {
                switch( mode ) {
                        case DIGIT:
                                if( isdigit( ch ) ) {
                                        buff[index] = ch;
                                        index++;
                                }
                                break;
                        case ALPHA:
                                if( isalpha( ch ) ) {
                                        buff[index] = ch;
                                        index++;
                                }
                                break;
                        case STRING:
                                if( isascii( ch ) ) {
                                        buff[index] = ch;
                                        index++;
                                }
                                break;
                        default:
                                /* this should not occur */
                                break;
                }
                ch = getchar();
        }
        buff[index] = 0x00;  /* null terminate input */
}

main() {
        char buffer[MAX];
        int number;

        printf("Please enter an integer\n");
        readinput( buffer, DIGIT, MAX );
        number = atoi( buffer );
        printf("The number you entered was %d\n", number );
}
```

Of course, there are improvements to be made. We can change *readinput* to return an integer value which represents the number of characters read. This would help in determining if data was actually entered. In the above program, it is not clear if the user actually entered any data (we could have checked to see if buffer was an empty array).

So lets now make the changes and see what the modified program looks like

```
/* version 1.1 */
#include <stdio.h>
#include <ctype.h>

#define MAX      80          /* maximum length of buffer          */
#define DIGIT     1          /* data will be read as digits 0-9   */
#define ALPHA     2          /* data will be read as alphabet A-Z */
#define STRING    3          /* data is read as ASCII             */

int readinput( char buff[], int mode, int limit ) {
        int ch, index = 0;
```

```
	ch = getchar();
	while( (ch != '\n') && (index < limit) ) {
		switch( mode ) {
			case DIGIT:
				if( isdigit( ch ) ) {
					buff[index] = ch;
					index++;
				}
				break;
			case ALPHA:
				if( isalpha( ch ) ) {
					buff[index] = ch;
					index++;
				}
				break;
			case STRING:
				if( isascii( ch ) ) {
					buff[index] = ch;
					index++;
				}
				break;
			default:
				/* this should not occur */
				break;
		}
		ch = getchar();
	}
	buff[index] = 0x00;  /* null terminate input */
	return index;
}

main() {
	char buffer[MAX];
	int number, digits = 0;

	while( digits == 0 ) {
		printf("Please enter an integer\n");
		digits = readinput( buffer, DIGIT, MAX );
		if( digits != 0 ) {
			number = atoi( buffer );
			printf("The number you entered was %d\n", number );
		}
	}
}
```

The second version is a much better implementation.

Controlling the cursor position

The following characters, placed after the \ character in a *printf()* statement, have the following effect.

```
	\b		backspace
	\f		form feed
```

```
        \n              new line
        \r              carriage return
        \t              horizontal tab
        \v              vertical tab
        \\              backslash
        \"              double quote
        \'              single quote
        \               line continuation
        \nnn            nnn = octal character value
        \0xnn           nn = hexadecimal value (some compilers only)

        printf("\007Attention, that was a beep!\n");
```

FORMATTERS FOR scanf()
The following characters, after the % character, in a scanf argument, have the following effect.

```
        d               read a decimal integer
        o               read an octal value
        x               read a hexadecimal value
        h               read a short integer
        l               read a long integer
        f               read a float value
        e               read a double value
        c               read a single character
        s               read a sequence of characters
        [...]           Read a character string. The characters inside the brackets
indicate
                        the allow-able characters that are to be contained in the
string. If any
                        other character is typed, the string is terminated. If the
first character
                        is a ^, the remaining characters inside the brackets
indicate that
                        typing them will terminate the string.
         *              this is used to skip input fields
```

Example of scanf() modifiers

```
        int number;
        char text1[30], text2[30];

        scanf("%s %d %*f %s", text1, &number, text2);
```

If the user response is,

```
        Hello 14 736.55 uncle sam
```

then

```
         text1 = hello, number = 14, text2 = uncle
```

and the next call to the scanf function will continue from where the last one left off, so if

```
        scanf("%s ", text2);
```

was the next call, then

```
         text2 = sam
```

PRINTING OUT THE ASCII VALUES OF CHARACTERS
Enclosing the character to be printed within single quotes will instruct the compiler to print out the Ascii value of the enclosed character.

```
        printf("The character A has a value of %d\n", 'A');
```

The program will print out the integer value of the character A.

# EXERCISE C20:
What would the result of the following operation be?

```
@       int c;
        c = 'a' + 1;
        printf("%c\n", c);

int c;
        c = 'a' + 1;
        printf("%c\n", c);

        The program adds one to the value 'a', resulting in the value 'b' as the
        value which is assigned to the variable c.
```

BIT OPERATIONS

C has the advantage of direct bit manipulation and the operations available are,

Operation	Operator	Comment	Value of Sum before	Value of sum after
AND	&	sum = sum & 2;	4	0
OR	|	sum = sum | 2;	4	6
Exclusive OR	^	sum = sum ^ 2;	4	6
1's Complement	~	sum = ~sum;	4	-5
Left Shift	<<	sum = sum << 2;	4	16
Right Shift	>>	sum = sum >> 2;	4	0

```
        /* Example program illustrating << and >> */
        #include <stdio.h>
```

```
main()
{
	int  n1 = 10, n2 = 20, i = 0;

	i = n2 << 4;  /* n2 shifted left four times */
	printf("%d\n", i);
	i = n1 >> 5;  /* n1 shifted right five times */
	printf("%d\n", i);
}
```

```
/* Example program using EOR operator  */
#include <stdio.h>

main()
{
	int  value1 = 2, value2 = 4;

	value1 ^= value2;
	value2 ^= value1;
	value1 ^= value2;
	printf("Value1 = %d, Value2 = %d\n", value1, value2);
}
```

```
/* Example program using AND operator  */
#include <stdio.h>

main()
{
	int  loop;

	for( loop = 'A'; loop <= 'Z'; loop++ )
		printf("Loop = %c, AND 0xdf = %c\n", loop, loop & 0xdf);
}
```

STRUCTURES

A Structure is a data type suitable for grouping data elements together. Lets create a new data structure suitable for storing the date. The elements or fields which make up the structure use the four basic data types. As the storage requirements for a structure cannot be known by the compiler, a definition for the structure is first required. This allows the compiler to determine the storage allocation needed, and also identifies the various sub-fields of the structure.

```
struct   date {
	int  month;
	int  day;
	int  year;
};
```

This declares a NEW data type called *date*. This date structure consists of three basic data elements, all of type integer. **This is a definition to the compiler.** It does not create any storage space and cannot be used as a variable. In essence, its a new data type keyword, like *int* and *char*, and can now be used to create variables. Other data structures may be defined as consisting of the same composition as the *date* structure,

```
        struct  date   todays_date;
```

defines a variable called *todays_date* to be of the same data type as that of the newly defined data type struct *date*.

ASSIGNING VALUES TO STRUCTURE ELEMENTS

To assign todays date to the individual elements of the structure *todays_date*, the statement

```
        todays_date.day = 21;
        todays_date.month = 07;
        todays_date.year = 1985;
```

is used. NOTE the use of the .element to reference the individual elements within *todays_date*.

```
        /* Program to illustrate a structure */
        #include <stdio.h>

        struct date {                   /* global definition of type date */
                int month;
                int day;
                int year;
        };

        main()
        {

                struct date  today;

                today.month = 10;
                today.day = 14;
                today.year = 1995;

                printf("Todays date is %d/%d/%d.\n", \
                        today.month, today.day, today.year );
        }
```

# EXERCISE C21:

Write a program in C that prompts the user for todays date, calculates tomorrows date, and displays the result. Use structures for todays date, tomorrows date, and an array to hold the days for each month of the year. Remember to change the month or year as necessary.

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

```
        struct date {
                int day, month, year;
        };

        int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        struct date today, tommorrow;

        void gettodaysdate( void );

        void gettodaysdate( void )
        {
                int valid = 0;

                while( valid == 0 ) {
                        printf("Enter in the current year (1990-1999)-->");
                        scanf("&d", &today.year);
                        if( (today.year < 1990) || (today.year > 1999) )
                                printf("\007Invalid year\n");
                        else
                                valid = 1;
                }
                valid = 0;
                while( valid == 0 ) {
                        printf("Enter in the current month (1-12)-->");
                        scanf("&d", &today.month);
                        if( (today.month < 1) || (today.month > 12) )
                                printf("\007Invalid month\n");
                        else
                                valid = 1;
                }
                valid = 0;
                while( valid == 0 ) {
                        printf("Enter in the current day (1-%d)-->",
days[today.month-1]);
                        scanf("&d", &today.day);
                        if( (today.day < 1) || (today.day > days[today.month-1]) )
                                printf("\007Invalid day\n");
                        else
                                valid = 1;
                }
        }

        main()
        {

                gettodaysdate();
                tommorrow = today;
                tommorrow.day++;
                if( tommorrow.day > days[tommorrow.month-1] ) {
                        tommorrow.day = 1;
                        tommorrow.month++;
                        if( tommorrow.month > 12 )
                                tommorrow.year++;
                }
                printf("Tommorrows date is %02d:%02d:%02d\n", \
                        tommorrow.day, tommorrow.month, tommorrow.year );
        }
```

```
/* TIME.C  Program updates time by 1 second using functions */
        #include <stdio.h>

        struct time {
            int hour, minutes, seconds;
        };

        void time time_update( struct time ); /* ANSI function prototype */

        /* function to update time by one second */
        void time_update( struct new_time )
        {
                ++new_time.seconds;
                if( new_time.seconds == 60) {
                        new_time.seconds = 0;
                        ++new_time.minutes;
                        if(new_time.minutes == 60) {
                                new_time.minutes = 0;
                                ++new_time.hour;
                                if(new_time.hour == 24)
                                        new_time.hour = 0;
                        }
                }
        }

        main()
        {
                void time time_update();
                struct time current_time;

                printf("Enter the time (hh:mm:ss):\n");
                scanf("%d:%d:%d", \

        &current_time.hour,&current_time.minutes,&current_time.seconds);
                time_update ( current_time);
                printf("The new time is %02d:%02d:%02d\n",current_time.hour, \
                        current_time.minutes, current_time.seconds);
        }
```

INITIALIZING STRUCTURES

This is similar to the initialization of arrays; the elements are simply listed inside a pair of braces, with each element separated by a comma. The structure declaration is preceded by the keyword *static*

```
        static struct date today = { 4,23,1998 };
```

ARRAYS OF STRUCTURES

Consider the following,

```
        struct  date  {
             int month, day, year;
        };
```
Lets now create an array called *birthdays* of the same data type as the structure *date*

```
        struct date birthdays[5];
```
This creates an array of 5 elements which have the structure of *date*.

```
        birthdays[1].month = 12;
        birthdays[1].day   = 04;
        birthdays[1].year  = 1998;
        --birthdays[1].year;
```

STRUCTURES AND ARRAYS
Structures can also contain arrays.

```
        struct month {
                int  number_of_days;
                char name[4];
        };

        static struct month this_month = { 31, "Jan" };

        this_month.number_of_days = 31;
        strcpy( this_month.name, "Jan" );
        printf("The month is %s\n", this_month.name );
```

Note that the array *name* has an extra element to hold the end of string nul character.

VARIATIONS IN DECLARING STRUCTURES
Consider the following,

```
        struct date {
             int month, day, year;
        } todays_date, purchase_date;
```
or another way is,

```
        struct date {
             int month, day, year;
        } todays_date = { 9,25,1985 };
```
or, how about an array of structures similar to date,

```
        struct date {
             int month, day, year;
        } dates[100];
```
Declaring structures in this way, however, prevents you from using the structure definition later in the program. The structure definition is thus bound to the variable name which follows the right brace of the structures definition.

# EXERCISE C22:

```
@       #include <stdio.h>

        struct  date  {        /* Global definition of date */
                int day, month, year;
        };

        main()
        {
                struct date dates[5];
                int i;

                for( i = 0; i < 5; ++i ) {
                        printf("Please enter the date (dd:mm:yy)" );
                        scanf("%d:%d:%d", &dates[i].day, &dates[i].month,
                                &dates[i].year );
                }
        }
```

STRUCTURES WHICH CONTAIN STRUCTURES

Structures can also contain structures. Consider where both a date and time structure are combined into a single structure called *date_time*, eg,

```
        struct date {
                int  month, day, year;
        };

        struct time {
                int  hours, mins, secs;
        };

        struct date_time {
                struct date sdate;
                struct time stime;
        };
```

This declares a structure whose elements consist of two other previously declared structures. Initialization could be done as follows,

```
        static struct date_time today = { { 2, 11, 1985 }, { 3, 3,33 } };
```

which sets the *sdate* element of the structure *today* to the eleventh of February, 1985. The *stime* element of the structure is initialized to three hours, three minutes, thirty-three seconds. Each item within the structure can be referenced if desired, eg,

```
        ++today.stime.secs;
        if( today.stime.secs == 60 ) ++today.stime.mins;
```

BIT FIELDS
Consider the following data elements defined for a PABX telephone system.

```
        flag = 1 bit
        off_hook = 1 bit
        status =  2 bits
```

In C, these can be defined as a structure, and the number of bits each occupy can be specified.

```
        struct packed_struct {
                unsigned int flag:1;
                unsigned int off_hook:1;
                unsigned int status:2;
        } packed_struct1;
```

The :1 following the variable *flag* indicates that flag occupies a single bit. The C compiler will assign all the above fields into a single word.

Assignment is as follows,

```
        packed_struct1.flag = 0;
        packed_struct1.status = 4;
        if( packed_struct1.flag )
                .............
```

Practise Exercise 9: Structures

1. Define a structure called *record* which holds an integer called *loop*, a character array of 5 elements called *word*, and a float called *sum*.

```
        struct record {
                int loop;
                char word[5];
                float sum;
        };
```

2. Declare a structure variable called *sample*, defined from a structure of type *struct record*.

```
        struct record sample;
```

3. Assign the value 10 to the field *loop* of the *sample* structure of type *struct record*.

```
        sample.loop = 10;
```

4. Print out (using printf) the value of the word array of the *sample* structure.

```
        printf("%s", sample.word );
```

5. Define a new structure called *birthdays*, whose fields are a structure of type *struct time* called *btime*, and a structure of type *struct date*, called *bdate*.

```
struct birthdays {
        struct time btime;
        struct date bdate;
};
```

DATA CONVERSION
The following functions convert between data types.

```
atof()          converts an ascii character array to a float
atoi()          converts an ascii character array to an integer
itoa()          converts an integer to a character array
```

Example

```
/* convert a string to an integer */
#include <stdio.h>
#include <stdlib.h>

char string[] = "1234";

main()
{
        int sum;
        sum = atoi( string );
        printf("Sum = %d\n", sum );
}
```

```
/* convert an integer to a string */
#include <stdio.h>
#include <stdlib.h>

main()
{
        int sum;
        char buff[20];

        printf("Enter in an integer ");
        scanf(" %d", &sum );
        printf( "As a string it is %s\n", itoa( sum, buff, 10 ) );
}
```

Note that itoa() takes three parameters,

- *the integer to b converted*
-
- *a character buffer into which the resultant string is stored*

-
- *a radix value (10=decimal, 16=hexadecimal)*
-

In addition, itoa() returns a pointer to the resultant string.

FILE INPUT/OUTPUT

To work with files, the library routines must be included into your programs. This is done by the statement,

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

as the first statement of your program.

USING FILES

- **Declare a variable of type FILE**
  To use files in C programs, you must declare a file variable to use. This variable must be of type **FILE**, and be declared as a pointer type.

  FILE is a predefined type. You declare a variable of this type as

  ```
          FILE  *in_file;
  ```

  This declares *infile* to be a pointer to a file.

- **Associate the variable with a file using fopen()**
  Before using the variable, it is associated with a specific file by using the *fopen()* function, which accepts the pathname for the file and the access mode (like reading or writing).
-
- `in_file = fopen( "myfile.dat", "r" );`

  In this example, the file **myfile.dat** in the current directory is opened for **read** access.

- **Process the data in the file**
  Use the appropriate file routines to process the data
- **When finished processing the file, close it**
  Use the *fclose()* function to close the file.
-
- `fclose( in_file );`

The following illustrates the *fopen* function, and adds testing to see if the file was opened successfully.

```
        #include <stdio.h>
        /* declares pointers to an input file, and the fopen function */
        FILE   *input_file, *fopen ();
```

```
        /* the pointer of the input file is assigned the value returned from the
fopen call. */
        /* fopen tries to open a file called datain for read only. Note that */
        /* "w" = write, and "a" = append.  */
        input_file = fopen("datain", "r");

        /* The pointer is now checked. If the file was opened, it will point to
the first */
        /* character of the file. If not, it will contain a NULL or 0. */
        if( input_file == NULL ) {
                printf("*** datain could not be opened.\n");
                printf("returning to dos.\n");
                exit(1);
        }
```

NOTE: Consider the following statement, which combines the opening of the file and its test to see if it was successfully opened into a single statement.

```
        if(( input_file = fopen ("datain", "r" )) == NULL ) {
                printf("*** datain could not be opened.\n");
                printf("returning to dos.\n");
                exit(1);
        }
```

INPUTTING/OUTPUTTING SINGLE CHARACTERS
Single characters may be read/written with files by use of the two functions, *getc()*, and *putc()*.

```
        int ch;

        ch = getc( input_file );   /*  assigns character to ch  */
```

The *getc()* also returns the value EOF (end of file), so

```
        while( (ch = getc( input_file )) != EOF )
                ......................
```

NOTE that the *putc/getc* are similar to *getchar/putchar* except that arguments are supplied specifying the I/O device.

```
        putc('\n', output_file ); /* writes a newline to output file */
```

CLOSING FILES

When the operations on a file are completed, it is closed before the program terminates. This allows the operating system to cleanup any resources or buffers associated with the file. The *fclose()* function is used to close the file and flush any buffers associated with the file.

```
        fclose( input_file );
        fclose( output_file );
```

COPYING A FILE

The following demonstrates copying one file to another using the functions we have just covered.

```
        #include <stdio.h>

        main()   /* FCOPY.C    */
        {
                char in_name[25], out_name[25];
                FILE *in_file, *out_file, *fopen ();
                int c;

                printf("File to be copied:\n");
                scanf("%24s", in_name);
                printf("Output filename:\n");
                scanf("%24s", out_name);

                in_file = fopen ( in_name, "r");

                if( in_file == NULL )
                        printf("Cannot open %s for reading.\n", in_name);
                else {
                        out_file = fopen (out_name, "w");
                        if( out_file == NULL )
                                printf("Can't open %s for writing.\n",out_name);
                        else {
                                while( (c = getc( in_file)) != EOF )
                                        putc (c, out_file);
                                putc (c, out_file);   /* copy EOF */
                                printf("File has been copied.\n");
                                fclose (out_file);
                        }
                        fclose (in_file);
                }
        }
```

TESTING FOR THE End Of File TERMINATOR (feof)

This is a built in function incorporated with the *stdio.h* routines. It returns 1 if the file pointer is at the end of the file.

```
        if( feof ( input_file ))
                printf("Ran out of data.\n");
```

THE fprintf AND fscanf STATEMENTS
These perform the same function as *printf* and *scanf*, but work on files. Consider,

```
        fprintf(output_file, "Now is the time for all..\n");
        fscanf(input_file, "%f", &float_value);
```

THE fgets AND fputs STATEMENTS
These are useful for reading and writing entire lines of data to/from a file. If *buffer* is a pointer to a character array and *n* is the maximum number of characters to be stored, then

```
        fgets (buffer, n, input_file);
```

will read an entire line of text (max chars = n) into *buffer* until the newline character or n=max, whichever occurs first. The function places a NULL character after the last character in the buffer. The function will be equal to a NULL if no more data exists.

```
        fputs (buffer, output_file);
```

writes the characters in *buffer* until a NULL is found. The NULL character is not written to the *output_file*.

NOTE: fgets does not store the newline into the buffer, fputs will append a newline to the line written to the output file.

Practise Exercise 9A: File Handling

1. Define an input file handle called *input_file*, which is a pointer to a type FILE.

```
        FILE *input_file;
```

2. Using *input_file*, open the file *results.dat* for read mode.

```
        input_file = fopen( "results.dat", "r" );
```

3. Write C statements which tests to see if *input_file* has opened the data file successfully. If not, print an error message and exit the program.

```
        if( input_file == NULL ) {
                printf("Unable to open file.\n");\
                exit(1);
        }
```

4. Write C code which will read a line of characters (terminated by a \n) from *input_file* into a character array called *buffer*. NULL terminate the buffer upon reading a \n.

```
int ch, loop = 0;

ch = fgetc( input_file );
while( (ch != '\n') && (ch != EOF) ) {
        buffer[loop] = ch;
        loop++;
        ch = fgetc( input_file );
}
buffer[loop] = NULL;
```

5. Close the file associated with *input_file*.

```
fclose( input_file );
```

File handling using open(), read(), write() and close()
The previous examples of file handling deal with File Control Blocks (FCB). Under MSDOS v3.x (or greater) and UNIX systems, file handling is often done using handles, rather than file control blocks.

Writing programs using handles ensures portability of source code between different operating systems. Using handles allows the programmer to treat the file as a stream of characters.

---

open()

```
#include <fcntl.h>
int  open(  char  *filename,  int  access,  int  permission  );
```

The available access modes are

```
O_RDONLY                O_WRONLY                O_RDWR
O_APPEND                O_BINARY                O_TEXT
```

The permissions are

```
S_IWRITE       S_IREAD S_IWRITE | S_IREAD
```

The *open()* function returns an integer value, which is used to refer to the file. If un- successful, it returns -1, and sets the global variable *errno* to indicate the error type.

---

read()

```
        #include  <fcntl.h>
        int  read(  int  handle,  void  *buffer,  int  nbyte );
```

The *read()* function attempts to read nbytes from the file associated with handle, and places the characters read into *buffer*. If the file is opened using O_TEXT, it removes carriage returns and detects the end of the file.

The function returns the number of bytes read. On end-of-file, 0 is returned, on error it returns -1, setting errno to indicate the type of error that occurred.

---

write()

```
        #include  <fcntl.h>
        int  write(  int  handle,  void  *buffer,  int  nbyte  );
```

The *write()* function attempts to write nbytes from *buffer* to the file associated with handle. On text files, it expands each LF to a CR/LF.

The function returns the number of bytes written to the file. A return value of -1 indicates an error, with errno set appropriately.

---

close()

```
        #include  <fcntl.h>
        int  close(  int  handle  );
```

The *close()* function closes the file associated with handle. The function returns 0 if successful, -1 to indicate an error, with errno set appropriately.

POINTERS
Pointers enable us to effectively represent complex data structures, to change values as arguments to functions, to work with memory which has been dynamically allocated, and to more concisely and efficiently deal with arrays. A pointer provides an indirect means of accessing the value of a particular data item. Lets see how pointers actually work with a simple example,

```
        int  count = 10, *int_pointer;
```
declares an integer *count* with a value of 10, and also an integer pointer called *int_pointer*. Note that the prefix * defines the variable to be of type pointer. To set up an indirect reference between *int_pointer* and *count*, the & prefix is used, ie,

```
        int_pointer = &count;
```

This assigns the memory address of *count* to int_pointer, not the actual value of *count* stored at that address.

POINTERS CONTAIN MEMORY ADDRESSES, NOT VALUES!

To reference the value of *count* using *int_pointer*, the * is used in an assignment, eg,

```
        x = *int_pointer;
```

Since *int_pointer* is set to the memory address of *count*, this operation has the effect of assigning the contents of the memory address pointed to by *int_pointer* to the variable *x*, so that after the operation variable *x* has a value of 10.

```
        #include <stdio.h>

        main()
        {
                int count  = 10, x, *int_pointer;

                /* this assigns the memory address of count to int_pointer  */
                int_pointer = &count;

                /* assigns the value stored at the address specified by int_pointer
to x */
                x = *int_pointer;

                printf("count = %d, x = %d\n", count, x);
        }
```

This however, does not illustrate a good use for pointers.

The following program illustrates another way to use pointers, this time with characters,

```
        #include <stdio.h>

        main()
        {
                char c = 'Q';
                char *char_pointer = &c;

                printf("%c %c\n", c, *char_pointer);

                c = 'Z';
                printf("%c %c\n", c, *char_pointer);
                *char_pointer = 'Y';
                /* assigns Y as the contents of the memory address specified by
char_pointer  */

                printf("%c %c\n", c, *char_pointer);
        }
```

# EXERCISE C23:
Determine the output of the pointer programs P1, P2, and P3.

```
        /* P1.C  illustrating pointers */
        #include <stdio.h>

        main()
        {
                int count  = 10, x, *int_pointer;

                /* this assigns the memory address of count to int_pointer  */
                int_pointer = &count;

                /* assigns the value stored at the address specified by int_pointer
to x */
                x = *int_pointer;

                printf("count = %d, x = %d\n", count, x);
        }
```

```
        /* P2.C  Further examples of pointers */
        #include <stdio.h>

        main()
        {
                char c = 'Q';
                char *char_pointer = &c;

                printf("%c %c\n", c, *char_pointer);

                c = '/';
                printf("%c %c\n", c, *char_pointer);
                *char_pointer = '(';
        /* assigns ( as the contents of the memory address specified by
char_pointer  */

                printf("%c %c\n", c, *char_pointer);
        }
```

@ Determine the output of the pointer programs P1, P2, and P3.

```
        /* P1.C  illustrating pointers */
        #include <stdio.h>

        main()
        {
                int count  = 10, x, *int_pointer;

                /* this assigns the memory address of count to int_pointer  */
                int_pointer = &count;

                /* assigns the value stored at the address specified by int_pointer 
to x */
                x = *int_pointer;

                printf("count = %d, x = %d\n", count, x);
        }
```

```
        count = 10, x = 10;
```

```
        /* P2.C  Further examples of pointers */
        #include <stdio.h>

        main()
        {
                char c = 'Q';
                char *char_pointer = &c;

                printf("%c %c\n", c, *char_pointer);

                c = '/';
                printf("%c %c\n", c, *char_pointer);
                *char_pointer = '(';
        /* assigns ( as the contents of the memory address specified by 
char_pointer  */

                printf("%c %c\n", c, *char_pointer);
        }
```

```
        Q Q
        / /
        ( (
```

```
        /* P3.C  Another program with pointers */
        #include <stdio.h>

        main()
        {
                int i1, i2, *p1, *p2;

                i1 = 5;
                p1 = &i1;
                i2 = *p1 / 2 + 10;
                p2 = p1;

                printf("i1 = %d, i2 = %d, *p1 = %d, *p2 = %d\n", i1, i2, *p1, *p2);
        }


        i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

Practise Exercise 10: Pointers

1. Declare a pointer to an integer called *address*.

```
int *address;
```

2. Assign the address of a float variable *balance* to the float pointer *temp*.

```
temp = &balance;
```

3. Assign the character value 'W' to the variable pointed to by the char pointer *letter*.

```
*letter = 'W';
```

4. What is the output of the following program segment?

```
int  count = 10, *temp; sum = 0;

temp = &count;
*temp = 20;
temp = &sum;
*temp = count;
printf("count = %d, *temp = %d, sum = %d\n", count, *temp, sum );

count = 20, *temp = 20, sum = 20
```

5. Declare a pointer to the text string "Hello" called *message*.

```
char *message = "Hello";
```

POINTERS AND STRUCTURES

Consider the following,

```
        struct date {
                int month, day, year;
        };

        struct date  todays_date, *date_pointer;

        date_pointer = &todays_date;

        (*date_pointer).day = 21;
        (*date_pointer).year = 1985;
        (*date_pointer).month = 07;

        ++(*date_pointer).month;
        if((*date_pointer).month == 08 )
                ......
```

Pointers to structures are so often used in C that a special operator exists. The structure pointer operator, the ->, permits expressions that would otherwise be written as,

```
        (*x).y
```

to be more clearly expressed as

```
        x->y
```

making the if statement from above program

```
        if( date_pointer->month == 08 )
                .....
```

```
        /* Program to illustrate structure pointers */
        #include <stdio.h>

        main()
        {
                struct date { int month, day, year; };
                struct date today, *date_ptr;

                date_ptr = &today;
                date_ptr->month = 9;
                date_ptr->day = 25;
                date_ptr->year = 1983;

                printf("Todays date is %d/%d/%d.\n", date_ptr->month, \
                        date_ptr->day, date_ptr->year % 100);
        }
```

So far, all that has been done could've been done without the use of pointers. Shortly, the real value of pointers will become apparent.

STRUCTURES CONTAINING POINTERS
Naturally, a pointer can also be a member of a structure.

```
        struct  int_pointers {
                int  *ptr1;
                int  *ptr2;
        };
```

In the above, the structure *int_pointers* is defined as containing two integer pointers, *ptr1* and *ptr2*. A variable of type struct int_pointers can be defined in the normal way, eg,

```
        struct  int_pointers  ptrs;
```

The variable *ptrs* can be used normally, eg, consider the following program,

```
        #include <stdio.h>
        main()   /* Illustrating structures containing pointers */
        {
                struct  int_pointers {  int   *ptr1, *ptr2;  };
                struct int_pointers  ptrs;
                int  i1 = 154, i2;

                ptrs.ptr1 =  &i1;
                ptrs.ptr2 =  &i2;
                (*ptrs).ptr2 =  -97;
                printf("i1 = %d, *ptrs.ptr1 = %d\n", i1, *ptrs.ptr1);
                printf("i2 = %d, *ptrs.ptr2 = %d\n", i2, *ptrs.ptr2);
        }
```

The following diagram may help to illustrate the connection,

```
        |------------|
        |  i1        |<---------------
        |------------|               |
        |  i2        |<-------       |
        |------------|        |      |
        |            |        |      |
        |------------|        |      |
        |  ptr1      |---------------
        |------------|        |          ptrs
        |  ptr2      |--------
        |------------|
```

POINTERS AND CHARACTER STRINGS
A pointer may be defined as pointing to a character string.

```
        #include <stdio.h>

        main()
        {
                char *text_pointer = "Good morning!";

                for( ; *text_pointer != '\0'; ++text_pointer)
                        printf("%c", *text_pointer);
        }
```

or another program illustrating pointers to text strings,

```
        #include <stdio.h>

        main()
        {
                static char *days[] = {"Sunday", "Monday", "Tuesday",
"Wednesday", \
                                                        "Thursday", "Friday",
"Saturday"};
                int i;

                for( i = 0; i < 6; ++i )
                        printf( "%s\n", days[i]);
        }
```

Remember that if the declaration is,

```
    char *pointer = "Sunday";
```

then the null character { '\0' } is automatically appended to the end of the text string. This means that %s may be used in a *printf* statement, rather than using a *for* loop and %c to print out the contents of the pointer. The %s will print out all characters till it finds the null terminator.

Practise Exercise 11: Pointers & Structures

1. Declare a pointer to a structure of type *date* called *dates*.

```
        struct date *dates;
```

2. If the above structure of type date comprises three integer fields, day, month, year, assign the value 10 to the field *day* using the *dates* pointer.

```
        dates->day = 10;
```

3. A structure of type *machine* contains two fields, an integer called *name*, and a char pointer called *memory*. Show what the definition of the structure looks like.

```
        |-----------| <---------
        |           | name       |
        |-----------|            | machine
        |           | memory     |
        |-----------| <---------
```

4. A pointer called *mpu641* of type machine is declared. What is the command to assign the value NULL to the field *memory*.

```
        mpu641->memory = (char *) NULL;
```

5. Assign the address of the character array *CPUtype* to the field *memory* using the pointer *mpu641*.

```
        mpu641->memory = CPUtype;
```

6. Assign the value 10 to the field *name* using the pointer *mpu641*.

```
        mpu641->name = 10;
```

7. A structure pointer *times* of type *time* (which has three fields, all pointers to integers, day, month and year respectively) is declared. Using the pointer *times*, update the field *day* to 10.

```
        *(times->day) = 10;
```

8. An array of pointers (10 elements) of type *time* (as detailed above in 7.), called *sample* is declared. Update the field *month* of the third array element to 12.

```
        *(sample[2]->month) = 12;
```

```
#include <stdio.h>

struct machine {
   int name;
   char *memory;
};

struct machine p1, *mpu641;

main()
{
   p1.name = 3;
   p1.memory = "hello";
   mpu641 = &p1;
   printf("name = %d\n", mpu641->name );
   printf("memory = %s\n", mpu641->memory );

   mpu641->name = 10;
   mpu641->memory = (char *) NULL;
   printf("name = %d\n", mpu641->name );
   printf("memory = %s\n", mpu641->memory );
}
```

```
#include <stdio.h>

struct time {
   int *day;
   int *month;
   int *year;
};

struct time t1, *times;

main()
{
   int d=5, m=12, y=1995;

   t1.day = &d;
   t1.month = &m;
   t1.year = &y;

   printf("day:month:year = %d:%d:%d\n", *t1.day, *t1.month, *t1.year );

   times = &t1;

   *(times->day) = 10;
   printf("day:month:year = %d:%d:%d\n", *t1.day, *t1.month, *t1.year );
}
```

Practise Exercise 11a: Pointers & Structures

Determine the output of the following program.

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

struct  record {
        char name[20];
        int id;
        float price;
};

void editrecord( struct record * );

void editrecord( struct record *goods )
{
        strcpy( goods->name, "Baked Beans" );
        goods->id = 220;
        (*goods).price = 2.20;
        printf("Name = %s\n", goods->name );
        printf("ID = %d\n", goods->id);
        printf("Price = %.2f\n", goods->price );
}

main()
{
        struct record item;

        strcpy( item.name, "Red Plum Jam");
        editrecord( &item );
        item.price = 2.75;
        printf("Name = %s\n", item.name );
        printf("ID = %d\n", item.id);
        printf("Price = %.2f\n", item.price );
}
```

1. Before call to editrecord()

```
@       item.name = "Red Plum Jam"
        item.id = 0
        item.price = 0.0
```

2. After return from editrecord()

```
@       item.name = "Baked Beans"
        item.id = 220
        item.price = 2.20
```

3. The final values of values, item.name, item.id, item.price

```
@       item.name = "Baked Beans"
```

```
        item.id = 220
        item.price = 2.75
```

C25: Examples on Pointer Usage
Determine the output of the following program.

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

struct  sample {
        char *name;
        int *id;
        float price;
};

static char product[]="Red Plum Jam";

main()
{
        int code = 312, number;
        char name[] = "Baked beans";
        struct sample item;

        item.name = product;
        item.id = &code;
        item.price = 2.75;
        item.name = name;
        number = *item.id;
        printf("Name = %s\n", item.name );
        printf("ID = %d\n", *item.id);
        printf("Price = %.2f\n", item.price );
}
```

```
@       Name = Baked Beans
        ID = 312
        Price = 2.75
```

C26: Examples on Pointer Usage
Determine the output of the following program.

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

struct  sample {
        char *name;
        int *id;
        float price;
};
```

```
static char  product[] = "Greggs Coffee";
static float price1 = 3.20;
static int   id = 773;

void printrecord( struct sample * );

void printrecord( struct sample *goods )
{
        printf("Name = %s\n", goods->name );
        printf("ID = %d\n", *goods->id);
        printf("Price = %.2f\n", goods->price );
        goods->name = &product[0];
        goods->id = &id;
        goods->price = price1;
}

main()
{
        int code = 123, number;
        char name[] = "Apple Pie";
        struct sample item;

        item.id = &code;
        item.price = 1.65;
        item.name = name;
        number = *item.id;
        printrecord( &item );
        printf("Name = %s\n", item.name );
        printf("ID = %d\n", *item.id);
        printf("Price = %.2f\n", item.price );
}
```

@ What are we trying to print out?

What does it evaluate to?

eg,

```
        printf("ID = %d\n", *goods->id);
        %d is an integer
                we want the value to be a variable integer type
        goods->id,
                what is id, its a pointer, so we mean contents of,
                        therefor we use *goods->id
                which evaluates to an integer type
```

```
Name = Apple Pie
ID = 123
Price = 1.65

Name = Greggs Coffee
ID = 773
```

```
Price = 3.20
```

File Handling Example

```
/* File handling example for PR101     */
/* processing an ASCII file of records */
/* Written by B. Brown, April 1994     */

/* process a goods file, and print out */
/* all goods where the quantity on     */
/* hand is less than or equal to the   */
/* re-order level.                     */

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

/* definition of a record of type goods */
struct goods {
   char  name[20];     /* name of product     */
   float price;        /* price of product    */
   int   quantity;     /* quantity on hand    */
   int   reorder;      /* re-order level      */
};

/* function prototypes */
void myexit( int );
void processfile( void );
void printrecord( struct goods );
int getrecord( struct goods * );

/* global data variables */
FILE *fopen(), *input_file;  /* input file pointer */

/* provides a tidy means to exit program gracefully */
void myexit( int exitcode )
{
   if( input_file != NULL )
      fclose( input_file );
   exit( exitcode );
}

/* prints a record */
void printrecord( struct goods record )
{
   printf("\nProduct name\t%s\n", record.name );
   printf("Product price\t%.2f\n", record.price );
   printf("Product quantity\t%d\n", record.quantity );
   printf("Product reorder level\t%d\n", record.reorder );
}

/* reads one record from inputfile into 'record', returns 1 for success */
```

```
int getrecord( struct goods *record )
{
   int loop = 0, ch;
   char buffer[40];

   ch = fgetc( input_file );
   /* skip to start of record */
   while( (ch == '\n') || (ch == ' ') && (ch != EOF) )
      ch = fgetc( input_file );
   if( ch == EOF ) return 0;

   /* read product name */
   while( (ch != '\n') && (ch != EOF)) {
      buffer[loop++] = ch;
      ch = fgetc( input_file );
   }
   buffer[loop] = 0;
   strcpy( record->name, buffer );
   if( ch == EOF ) return 0;

   /* skip to start of next field */
   while( (ch == '\n') || (ch == ' ') && (ch != EOF) )
      ch = fgetc( input_file );
   if( ch == EOF ) return 0;

   /* read product price */
   loop = 0;
   while( (ch != '\n') && (ch != EOF)) {
      buffer[loop++] = ch;
      ch = fgetc( input_file );
   }
   buffer[loop] = 0;
   record->price = atof( buffer );
   if( ch == EOF ) return 0;

   /* skip to start of next field */
   while( (ch == '\n') || (ch == ' ') && (ch != EOF) )
      ch = fgetc( input_file );
   if( ch == EOF ) return 0;

   /* read product quantity */
   loop = 0;
   while( (ch != '\n') && (ch != EOF)) {
      buffer[loop++] = ch;
      ch = fgetc( input_file );
   }
   buffer[loop] = 0;
   record->quantity = atoi( buffer );
   if( ch == EOF ) return 0;

   /* skip to start of next field */
   while( (ch == '\n') || (ch == ' ') && (ch != EOF) )
      ch = fgetc( input_file );
   if( ch == EOF ) return 0;

   /* read product reorder level */
   loop = 0;
   while( (ch != '\n') && (ch != EOF)) {
      buffer[loop++] = ch;
      ch = fgetc( input_file );
   }
   buffer[loop] = 0;
   record->reorder = atoi( buffer );
   if( ch == EOF ) return 0;

   return 1;  /* signify record has been read successfully */
}

/* processes file for records */
void processfile( void )
{
   struct goods record;   /* holds a record read from inputfile */

   while( ! feof( input_file )) {
      if( getrecord( &record ) == 1 ) {
         if( record.quantity <= record.reorder )
            printrecord( record );
      }
      else myexit( 1 );  /* error getting record */
   }
}

main()
{
   char filename[40];     /* name of database file */

   printf("Example Goods Re-Order File Program\n");
   printf("Enter database file ");
   scanf(" %s", filename );
   input_file = fopen( filename, "rt" );
   if( input_file == NULL ) {
      printf("Unable to open datafile %s\n", filename );
      myexit( 1 );
   }
   processfile();
   myexit( 0 );
}
```

Please obtain the data file for this example from your tutor, or via **ftp**.

File Handling Example
The data file for this exercise looks like,

```
baked beans
1.20
10
5

greggs coffee
2.76
5
```

```
10

walls ice-cream
3.47
5
5

cadburys chocs
4.58
12
10
```

LINKED LISTS

A linked list is a complex data structure, especially useful in systems or applications programming. A linked list is comprised of a series of nodes, each node containing a data element, and a pointer to the next node, eg,

```
         --------            --------
        |  data  |      --->|  data  |
        |--------|     |    |--------|
        | pointer|----      | pointer| ---> NULL
         --------            --------
```

A structure which contains a data element and a pointer to the next node is created by,

```
        struct list {
                int   value;
                struct list  *next;
        };
```

This defines a new data structure called *list* (actually the definition of a node), which contains two members. The first is an integer called *value*. The second is called *next*, which is a pointer to another list structure (or node). Suppose that we declare two structures to be of the same type as list, eg,

```
        struct list  n1, n2;
```

The next pointer of structure *n1* may be set to point to the *n2* structure by

```
        /* assign address of first element in n2 to the pointer next of the n1
structure  */
        n1.next = &n2;
```

which creates a link between the two structures.

```
        /* LLIST.C    Program to illustrate linked lists */
        #include <stdio.h>
```

```
struct list {
        int         value;
        struct list *next;
};

main()
{
        struct list n1, n2, n3;
        int          i;

        n1.value = 100;
        n2.value = 200;
        n3.value = 300;
        n1.next = &n2;
        n2.next = &n3;
        i = n1.next->value;
        printf(%d\n", n2.next->value);
}
```

Not only this, but consider the following

```
n1.next = n2.next      /* deletes n2  */
n2_3.next = n2.next;   /* adds struct n2_3 */
n2.next = &n2_3;
```

In using linked list structures, it is common to assign the value of 0 to the last pointer in the list, to indicate that there are no more nodes in the list, eg,

```
n3.next = 0;
```

Traversing a linked list

```
/* Program to illustrate traversing a list */
#include <stdio.h>
struct list {
        int         value;
        struct list *next;
};

main()
{
        struct list n1, n2, n3, n4;
        struct list *list_pointer = &n1;

        n1.value = 100;
        n1.next = &n2;
        n2.value = 200;
        n2.next = &n3;
        n3.value = 300;
        n3.next = &n4;
        n4.value = 400;
```

```
		n4.next = 0;

		while( list_pointer != 0 )  {
			printf("%d\n", list_pointer->value);
			list_pointer = list_pointer->next;
		}
	}
```

This program uses a pointer called *list_pointer* to cycle through the linked list.

Practise Exercise 12: Lists

. Define a structure called *node*, which contains an integer element called *data*, and a pointer to a structure of type *node* called *next_node*.

2. Declare three structures called *node1*, *node2*, *node3*, of type *node*.

3. Write C statements which will link the three nodes together, with node1 at the head of the list, node2 second, and node3 at the tail of the list. Assign the value NULL to node3.next to signify the end of the list.

4. Using a pointer *list*, of type *node*, which has been initialised to the address of *node1*, write C statements which will cycle through the list and print out the value of each nodes data field.

5. Assuming that pointer *list* points to *node2*, what does the following statement do?

```
	list->next_node = (struct node *) NULL;
```

6. Assuming the state of the list is that as in 3., write C statements which will insert a new node *node1a* between node1 and node2, using the pointer *list* (which is currently pointing to node1). Assume that a pointer *new_node* points to node node1a.

7. Write a function called *delete_node*, which accepts a pointer to a list, and a pointer to the node to be deleted from the list, eg

```
	void  delete_node(  struct  node  *head,  struct  node  *delnode );
```

8. Write a function called *insert_node*, which accepts a pointer to a list, a pointer to a new node to be inserted, and a pointer to the node after which the insertion takes place, eg

```
        void insert_node( struct node *head, struct node *newnode, struct node 
*prevnode );
```

Answers

Practise Exercise 12: Lists

1. Define a structure called *node*, which contains an integer element called *data*, and a pointer to a structure of type *node* called *next_node*.

```
        struct node {
                int data;
                struct node *next_node;
        };
```

2. Declare three structures called *node1*, *node2*, *node3*, of type *node*.

```
        struct node node1, node3, node3;
```

3. Write C statements which will link the three nodes together, with node1 at the head of the list, node2 second, and node3 at the tail of the list. Assign the value NULL to node3.next to signify the end of the list.

```
        node1.next_node = &node2;
        node2.next_node = &node3;
        node3.next_node = (struct node *) NULL;
```

4. Using a pointer *list*, of type *node*, which has been initialised to the address of *node1*, write C statements which will cycle through the list and print out the value of each nodes data field.

```
        while( list != NULL ) {
                printf("%d\n", list->data );
                list = list->next_node;
        }
```

5. Assuming that pointer *list* points to *node2*, what does the following statement do?

```
        list->next_node = (struct node *) NULL;

        The statement writes a NULL into the next_node pointer, making node2 the 
end of
```

```
        the list, thereby erasing node3 from the list.
```

6. Assuming the state of the list is that as in 3., write C statements which will insert a new node *node1a* between node1 and node2, using the pointer *list* (which is currently pointing to node1). Assume that a pointer *new_node* points to node node1a.

```
        new_node.next_node = list.next_node;
        list.next_node = new_node;
```

7. Write a function called *delete_node*, which accepts a pointer to a list, and a pointer to the node to be deleted from the list, eg

```
        void  delete_node(  struct  node  *head,  struct  node  *delnode );


        void  delete_node( struct node *head, struct node *delnode )
        {
                struct node *list;

                list = head;
                while( list->next != delnode ) {
                        list = list->node;

                list->next = delnode->next;
        }
```

8. Write a function called *insert_node*, which accepts a pointer to a list, a pointer to a new node to be inserted, and a pointer to the node after which the insertion takes place, eg

```
        void insert_node( struct node *head, struct node *newnode, struct node
*prevnode );


        void insert_node( struct node *head, struct node *newnode, struct node
*prevnode )
        {
                struct node *list;

                list = head;
                while( list != prevnode )
                        list = list->next;

                newnode->next = list->next;
                list->next = newnode;
        }
```

DYNAMIC MEMORY ALLOCATION (CALLOC, SIZEOF, FREE)

It is desirable to dynamically allocate space for variables at runtime. It is wasteful when dealing with array type structures to allocate so much space when declared, eg,

```
        struct client clients[100];
```

This practice may lead to memory contention or programs crashing. A far better way is to allocate space to clients when needed.

The C programming language allows users to dynamically allocate and deallocate memory when required. The functions that accomplish this are *calloc()*, which allocates memory to a variable, *sizeof*, which determines how much memory a specified variable occupies, and *free()*, which deallocates the memory assigned to a variable back to the system.

SIZEOF

The sizeof() function returns the memory size of the requested variable. This call should be used in conjunction with the *calloc()* function call, so that only the necessary memory is allocated, rather than a fixed size. Consider the following,

```
        struct date {
                int hour, minute, second;
        };

        int x;

        x = sizeof( struct date );
```

x now contains the information required by *calloc()* so that it can allocate enough memory to contain another structure of type *date*.

CALLOC

This function is used to allocate storage to a variable whilst the program is running. The function takes two arguments that specify the number of elements to be reserved, and the size of each element (obtained from *sizeof*) in bytes. The function returns a character pointer (void in ANSI C) to the allocated storage, which is initialized to zero's.

```
        struct date *date_pointer;

        date_pointer = (struct date *)  calloc( 10, sizeof(struct date) );
```

The (struct date *) is a type cast operator which converts the pointer returned from *calloc* to a character pointer to a structure of type *date*. The above function call will allocate size for ten such structures, and *date_pointer* will point to the first in the chain.

FREE

When the variables are no longer required, the space which was allocated to them by *calloc* should be returned to the system. This is done by,

```
        free( date_pointer );
```

Other C calls associated with memory are,

```
        alloc           allocate a block of memory from the heap
        malloc          allocate a block of memory, do not zero out
        zero            zero a section of memory
        blockmove       move bytes from one location to another
```

Other routines may be included in the particular version of the compiler you may have, ie, for MS-DOS v3.0,

```
        memccpy         copies characters from one buffer to another
        memchr          returns a pointer to the 1st occurrence of a
                        designated character searched for
        memcmp          compares a specified number of characters
        memcpy          copies a specified number of characters
        memset          initialise a specified number of bytes with a given
character
        movedata        copies characters
```

EXAMPLE OF DYNAMIC ALLOCATION

```
/* linked list example, pr101, 1994 */
#include <string.h>
#include <alloc.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <conio.h>

/* definition of a node */
```

```
struct node {
    char data[20];
    struct node *next;
};


struct node * initialise( void );
void freenodes( struct node * );
int insert( struct node * );
void delete( struct node *, struct node * );
void list( struct node * );
void menu( struct node *, struct node * );
void readline( char [] );



void readline( char buff[] )
{
    int ch, loop = 0;

    ch = getche();
    while( ch != '\r' ) {
       buff[loop] = ch;
       loop++;
       ch = getche();
    }
    buff[loop] = 0;
}

struct node * initialise( void )
{
    return( (struct node *) calloc(1, sizeof( struct node *) ));
}

/* free memory allocated for node */
void freenodes( struct node *headptr )
{
    struct node *temp;
    while( headptr ) {
       temp = headptr->next;
       free( headptr );
       headptr = temp;
    }
}

/* insert a new node after nodeptr, return 1 = success */
int insert( struct node *nodeptr )
{
    char buffer[20];
    struct node *newptr;

    newptr = initialise(); /* allocate a new node */
    if( newptr == NULL ) {
        return 0;
    }
    else {                 /* fill in its data and add to the list */
        newptr->next = nodeptr->next;
        nodeptr->next = newptr;
        nodeptr = newptr;
        printf("\nEnter data --->");
        readline( buffer );
        strcpy( nodeptr->data, buffer );
    }
    return 1;
}

/* delete a node from list */
void delete( struct node *headptr, struct node *nodeptr )
{
    struct node *deletepointer, *previouspointer;
    char buffer[20];

    deletepointer = headptr->next;
    previouspointer = headptr;
    /* find the entry */
    printf("\nEnter name to be deleted --->");
    readline( buffer );
    while( deletepointer ) {
        if( strcmp( buffer, deletepointer->data ) == 0 ) {
            /* delete node pointed to by delete pointer */
            previouspointer->next = deletepointer->next;
            break;
        }
        else {
            /* goto next node in list */
            deletepointer = deletepointer->next;
            previouspointer = previouspointer->next;
        }
    }
    /* did we find it? */
    if( deletepointer == NULL )
        printf("\n\007Error, %s not found or list empty\n", buffer);
    else {
        free( deletepointer );
        /* adjust nodeptr to the last node in list */
        nodeptr = headptr;
        while( nodeptr->next != NULL )
            nodeptr = nodeptr->next;
    }
}

/* print out the list */
void list( struct node *headptr )
{
    struct node *listpointer;

    listpointer = headptr->next;
    if( listpointer == NULL )
        printf("\nThe list is empty.\n");
    else {
        while( listpointer ) {
            printf("Name : %20s\n", listpointer->data );
            listpointer = listpointer->next;
        }
    }
}

/* main menu system */
void menu( struct node *headp, struct node *nodep )
{
    int menuchoice = 1;
    char buffer[20];

    while( menuchoice != 4 ) {
       printf("1  insert a node\n");
       printf("2  delete a node\n");
       printf("3  list nodes\n");
       printf("4  quit\n");
       printf("Enter choice -->");
       readline( buffer );
       menuchoice = atoi( buffer );
       switch( menuchoice ) {
           case 1 : if( insert( nodep ) == 0 )
                        printf("\n\007Insert failed.\n");
                    break;
           case 2 : delete( headp, nodep );  break;
           case 3 : list( headp );    break;
           case 4 : break;
           default : printf("\n\007Invalid option\n"); break;
       }
    }
}

main()
{
    struct node *headptr, *nodeptr;
    headptr = initialise();
    nodeptr = headptr;
    headptr->next = NULL;
    menu( headptr, nodeptr );
    freenodes( headptr );
}
```

Another Linked List Example

```
/* linked list example */
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

/* function prototypes */
struct node * initnode( char *, int );
void printnode( struct node * );
void printlist( struct node * );
void add( struct node * );
struct node * searchname( struct node *, char * );
void deletenode( struct node * );
void insertnode( struct node * );
void deletelist( struct node * );

/* definition of a data node for holding student information */
struct node {
   char name[20];
   int  id;
   struct node *next;
};

/* head points to first node in list, end points to last node in list */
/* initialise both to NULL, meaning no nodes in list yet */
struct node *head = (struct node *) NULL;
struct node *end = (struct node *) NULL;

/* this initialises a node, allocates memory for the node, and returns   */
/* a pointer to the new node. Must pass it the node details, name and id */
struct node * initnode( char *name, int id )
{
   struct node *ptr;
   ptr = (struct node *) calloc( 1, sizeof(struct node ) );
   if( ptr == NULL )                       /* error allocating node?      */
       return (struct node *) NULL;        /* then return NULL, else      */
   else {                                  /* allocated node successfully */
       strcpy( ptr->name, name );          /* fill in name details        */
       ptr->id = id;                       /* copy id details             */
       return ptr;                         /* return pointer to new node  */
   }
}

/* this prints the details of a node, eg, the name and id                 */
/* must pass it the address of the node you want to print out             */
void printnode( struct node *ptr )
{
   printf("Name ->%s\n", ptr->name );
   printf("ID   ->%d\n", ptr->id );
}

/* this prints all nodes from the current address passed to it. If you    */
/* pass it 'head', then it prints out the entire list, by cycling through */
/* each node and calling 'printnode' to print each node found             */
void printlist( struct node *ptr )
{
   while( ptr != NULL )           /* continue whilst there are nodes left */
   {
      printnode( ptr );           /* print out the current node           */
      ptr = ptr->next;            /* goto the next node in the list       */
   }
}

/* this adds a node to the end of the list. You must allocate a node and  */
/* then pass its address to this function                                 */
void add( struct node *new )  /* adding to end of list */
{
   if( head == NULL )       /* if there are no nodes in list, then         */
       head = new;          /* set head to this new node                   */
   end->next = new;         /* link in the new node to the end of the list */
   new->next = NULL;        /* set next field to signify the end of list   */
   end = new;               /* adjust end to point to the last node        */
}

/* search the list for a name, and return a pointer to the found node     */
/* accepts a name to search for, and a pointer from which to start. If    */
/* you pass the pointer as 'head', it searches from the start of the list */
struct node * searchname( struct node *ptr, char *name )
{
    while( strcmp( name, ptr->name ) != 0 ) {    /* whilst name not found */
       ptr = ptr->next;                          /* goto the next node    */
       if( ptr == NULL )                         /* stop if we are at the */
          break;                                 /* of the list           */
    }
    return ptr;                                  /* return a pointer to   */
}                                                /* found node or NULL    */

/* deletes the specified node pointed to by 'ptr' from the list           */
void deletenode( struct node *ptr )
{
   struct node *temp, *prev;
   temp = ptr;    /* node to be deleted */
   prev = head;   /* start of the list, will cycle to node before temp    */

   if( temp == prev ) {                    /* are we deleting first node  */
       head = head->next;                  /* moves head to next node     */
       if( end == temp )                   /* is it end, only one node?   */
          end = end->next;                 /* adjust end as well          */
       free( temp );                       /* free space occupied by node */
   }
   else {                                  /* if not the first node, then */
       while( prev->next != temp ) {       /* move prev to the node before*/
           prev = prev->next;              /* the one to be deleted       */
       }
       prev->next = temp->next;            /* link previous node to next  */
       if( end == temp )                   /* if this was the end node,   */
           end = prev;                     /* then reset the end pointer  */
       free( temp );                       /* free space occupied by node */
   }
}

/* inserts a new node, uses name field to align node as alphabetical list */
/* pass it the address of the new node to be inserted, with details all   */
/* filled in                                                              */
void insertnode( struct node *new )
{
   struct node *temp, *prev;                /* similar to deletenode      */

   if( head == NULL ) {                     /* if an empty list,          */
       head = new;                          /* set 'head' to it           */
       end = new;
       head->next = NULL;                   /* set end of list to NULL    */
       return;                              /* and finish                 */
   }
   temp = head;                            /* start at beginning of list */
                  /* whilst currentname < newname to be inserted then */
   while( strcmp( temp->name, new->name) < 0 ) {
          temp = temp->next;               /* goto the next node in list */
          if( temp == NULL )               /* dont go past end of list   */
              break;
   }

   /* we are the point to insert, we need previous node before we insert  */
   /* first check to see if its inserting before the first node!          */
   if( temp == head ) {
      new->next = head;            /* link next field to original list   */
      head = new;                  /* head adjusted to new node          */
   }
   else {      /* okay, so its not the first node, a different approach    */
      prev = head;   /* start of the list, will cycle to node before temp */
      while( prev->next != temp ) {
          prev = prev->next;
      }
      prev->next = new;            /* insert node between prev and next  */
      new->next = temp;
      if( end == prev )            /* if the new node is inserted at the */
         end = new;                /* end of the list the adjust 'end'   */
   }
}

/* this deletes all nodes from the place specified by ptr                 */
/* if you pass it head, it will free up entire list                       */
void deletelist( struct node *ptr )
{
   struct node *temp;

   if( head == NULL ) return;   /* dont try to delete an empty list       */

   if( ptr == head ) {      /* if we are deleting the entire list         */
       head = NULL;         /* then reset head and end to signify empty   */
       end = NULL;          /* list                                       */
   }
   else {
       temp = head;          /* if its not the entire list, readjust end  */
       while( temp->next != ptr )          /* locate previous node to ptr  */
           temp = temp->next;
       end = temp;                         /* set end to node before ptr   */
   }

   while( ptr != NULL ) {   /* whilst there are still nodes to delete     */
      temp = ptr->next;     /* record address of next node                */
      free( ptr );          /* free this node                             */
      ptr = temp;           /* point to next node to be deleted           */
   }
}

/* this is the main routine where all the glue logic fits                 */
main()
{
   char name[20];
```

```
int id, ch = 1;
struct node *ptr;

clrscr();
while( ch != 0 ) {
   printf("1 add a name \n");
   printf("2 delete a name \n");
   printf("3 list all names \n");
   printf("4 search for name \n");
   printf("5 insert a name \n");
   printf("0 quit\n");
   scanf("%d", &ch );
   switch( ch )
   {
       case 1:  /* add a name to end of list */
                printf("Enter in name -- ");
                scanf("%s", name );
                printf("Enter in id -- ");
                scanf("%d", &id );
                ptr = initnode( name, id );
                add( ptr );
                break;
       case 2:  /* delete a name */
                printf("Enter in name -- ");
                scanf("%s", name );
                ptr = searchname( head, name );
                if( ptr ==NULL ) {
                    printf("Name %s not found\n", name );
                }
                else
                   deletenode( ptr );
                break;

       case 3:  /* list all nodes */
                printlist( head );
                break;

       case 4:  /* search and print name */
                printf("Enter in name -- ");
                scanf("%s", name );
                ptr = searchname( head, name );
                if( ptr ==NULL ) {
                    printf("Name %s not found\n", name );
                }
                else
                   printnode( ptr );
                break;
       case 5:  /* insert a name in list */
                printf("Enter in name -- ");
                scanf("%s", name );
                printf("Enter in id -- ");
                scanf("%d", &id );
                ptr = initnode( name, id );
                insertnode( ptr );
                break;

   }
   }
   deletelist( head );
}
```

PREPROCESSOR STATEMENTS
The *define* statement is used to make programs more readable, and allow the inclusion of macros. Consider the following examples,

```
        #define TRUE     1    /* Do not use a semi-colon , # must be first 
character on line */
        #define FALSE    0
        #define NULL     0
        #define AND      &
        #define OR       |
        #define EQUALS  ==

                       game_over = TRUE;
                       while( list_pointer != NULL )
                               ................
```

Macros

Macros are inline code which are substituted at compile time. The definition of a macro, which accepts an argument when referenced,

```
        #define  SQUARE(x)  (x)*(x)

        y = SQUARE(v);
```

In this case, *v* is equated with *x* in the macro definition of *square*, so the variable *y* is assigned the square of *v*. The brackets in the macro definition of *square* are necessary for correct evaluation. The expansion of the macro becomes

```
        y = (v) * (v);
```

Naturally, macro definitions can also contain other macro definitions,

```
        #define IS_LOWERCASE(x)  (( (x)>='a') && ( (x) <='z') )
        #define TO_UPPERCASE(x)  (IS_LOWERCASE (x)?(x)-'a'+'A':(x))

        while(*string) {
                *string = TO_UPPERCASE(*string);
                ++string;
        }
```

CONDITIONAL COMPILATIONS

These are used to direct the compiler to compile/or not compile the lines that follow

```
        #ifdef  NULL
        #define NL 10
        #define SP 32
        #endif
```

In the preceding case, the definition of NL and SP will only occur if NULL has been defined prior to the compiler encountering the #ifdef NULL statement. The scope of a definition may be limited by

```
        #undef NULL
```

This renders the identification of NULL invalid from that point onwards in the source file.

typedef

This statement is used to classify existing C data types, eg,

```
        typedef  int counter;  /* redefines counter as an integer */
        counter j, n;          /* counter now used to define j and n as integers
*/

        typedef struct {
                int month, day, year;
        } DATE;

        DATE  todays_date;     /* same as struct date todays_date */
```

ENUMERATED DATA TYPES

Enumerated data type variables can only assume values which have been previously declared.

```
        enum month { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov,
dec };
        enum month this_month;

        this_month = feb;
```

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *this_month* is declared to be of the same type as month, then is assigned the value associated with feb. This_month cannot be assigned any values outside those specified in the initialization list for the declaration of month.

```
#include <stdio.h>

        main()
        {
                char *pwest = "west",*pnorth = "north", *peast="east", *psouth = 
"south";
                enum location { east=1, west=2, south=3, north=4};
                enum location direction;

                direction = east;

                if( direction == east )
                        printf("Cannot go %s\n", peast);
        }
```

The variables defined in the enumerated variable *location* should be assigned initial values.

UNIONS
This is a special data type which looks similar to a structure, but is very different. The declaration is,

```
        union  mixed {
                char  letter;
                float radian;
                int   number;
        };

        union mixed all;
```

The first declaration consists of a union of type mixed, which consists of a char, float, or int variable. NOTE that it can be ONLY ONE of the variable types, they cannot coexist.

This is due to the provision of a single memory address which is used to store the largest variable, unlike the arrangement used for structures.

Thus the variable *all* can only be a character, a float or an integer at any one time. The C language keeps track of what *all* actually is at any given moment, but does not provide a check to prevent the programmer accessing it incorrectly.

DECLARING VARIABLES TO BE REGISTER BASED
Some routines may be time or space critical. Variables can be defined as being register based by the following declaration,

```
        register int index;
```

DECLARING VARIABLES TO BE EXTERNAL
Here variables may exist in separately compiled modules, and to declare that the variable is external,

```
        extern  int  move_number;
```

This means that the data storage for the variable *move_number* resides in another source module, which will be linked with this module to form an executable program. In using a variable across a number of independently compiled modules, space should be allocated in only one module, whilst all other modules use the extern directive to access the variable.

NULL STATEMENTS
These are statements which do not have any body associated with them.

```
        /* sums all integers in array a containing n elements and initializes */
        /* two variables at the start of the for loop */
        for( sum = 0, i = 0; i < n; sum += a[i++] )
                ;
```

```
        /* Copies characters from standard input to standard output until EOF is
reached  */
        for( ; (c = getchar ()) != EOF; putchar (c))
                ;
```

STRINGS
Consider the following,

```
        char  *text_pointer = "Hello said the man.";
```

This defines a character pointer called *text_pointer* which points to the start of the text string 'Hello said the man'. This message could be printed out by

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

text_pointer holds the memory address of where the message is located in memory.

Lets append two strings together by using arrays.

```
        #include <stdio.h>

        main()
        {
                static char string1[]={'H','e','l','l','o',' ' };
                static char string2[]={'s','a','i','d',' ','t','h','e','
','m','a','n','.'  };
                char  string3[25];
                int string_length1 = 6, string_length2 = 13, n;

                for( n = 0; n < string_length1; ++n )
                        string3[n] = string1[n];

                for( n = 0; n < string_length2; ++n )
                        string3[n + string_length1] = string2[n];

                for(n = 0; n < (stringlength1+string_length2); ++n)
                        printf("%c", string3[n]);
        }
```

Strings continued
There are times that the length of a string may not be known. Consider the following improvements by terminating each string with a null character.

```
        #include <stdio.h>

        main()
        {
                static char  string1[] = "Bye Bye ";
                static char  string2[] = "love.";
                char  string3[25];
                int  n = 0, n2;

                for( ; string1[n] != '\0'; ++n )
                        string3[n] = string1[n];
                n2 = n;  n = 0;

                for( ; string2[n] != '\0'; ++n )
                        string3[n2 + n] = string2[n];

                n2 += n;

                for(n = 0; n < n2 ; ++n)
                        printf("%c", string3[n]);
        }
```

```
Minor modification to above program is,

        string3[n2 + n] = '\0';
        printf("%s", string3);
```

FURTHER IMPROVEMENTS by using POINTERS
The previous program still required the use of variables to keep track of string lengths. Implementing concatenation by the use of pointers eliminates this, eg,

```
        #include <stdio.h>

        void concat( char *, char *, char * );

        /* this functions copies the strings a and b to the destination string c
*/
        void concat( char *a, char *b, char *c)
        {
                while( *a )  {          /* while( *c++ = *a++ );  */
                        *c = *a; ++a; ++c;
                }
                while( *b )  {
                        *c = *b; ++b; ++c;
                }
                *c = '\0';
        }

        main()
        {
                static char string1[] = "Bye Bye ";
                static char string2[] = "love.";
                char string3[20];

                concat( string1, string2, string3);
                printf("%s\n", string3);
        }
```

USING strcat IN THE LIBRARY ROUTINE string.h
The following program illustrates using the supplied function resident in the appropriate library file. *strcat()* concatenates one string onto another and returns a pointer to the concatenated string.

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

        main()
        {
                static char string1[] = "Bye Bye ";
                static char string2[] = "love.";
                char *string3;

                string3 = strcat ( string1, string2 );
                printf("%s\n", string3);
}
```

COMMAND LINE ARGUMENTS
It is possible to pass arguments to C programs when they are executed. The brackets which follow main are used for this purpose. *argc* refers to the number of arguments passed, and *argv[]* is a pointer array which points to each argument which is passed to main. A simple example follows, which checks to see if a single argument is supplied on the command line when the program is invoked.

```
        #include <stdio.h>

        main( int argc, char *argv[] )
        {
                if( argc == 2 )
                        printf("The argument supplied is %s\n", argv[1]);
                else if( argc > 2 )
                        printf("Too many arguments supplied.\n");
                else
                        printf("One argument expected.\n");
        }
```

Note that **argv[0]* is the name of the program invoked, which means that **argv[1]* is a pointer to the first argument supplied, and **argv[n]* is the last argument. If no arguments are supplied, *argc* will be one. Thus for n arguments, *argc* will be equal to n + 1. The program is called by the command line,

```
         myprog  argument1
```

EXERCISE C27
Rewrite the program which copies files, ie, FCOPY.C to accept the source and destination filenames from the command line. Include a check on the number of arguments passed.

Answer

Rewrite the program which copies files, ie, FCOPY.C to accept the source and destination filenames from the command line. Include a check on the number of arguments passed.

```
        #include <stdio.h>

        main( int argc, char *argv[])
        {
                FILE *in_file, *out_file, *fopen();
                int c;

                if( argc != 3 )
                {
                        printf("Incorrect, format is FCOPY source dest\n");
                        exit(2);
                }
                in_file = fopen( argv[1], "r");
                if( in_file == NULL )  printf("Cannot open %s for reading\n",
argv[1]);
                else {
                        out_file = fopen( argv[2], "w");
                        if ( out_file == NULL )
                                printf("Cannot open %s for writing\n", argv[2]);
                        else {
                                printf("File copy program, copying %s to %s\n",
argv[1],  argv[2]);
                                while ( (c=getc( in_file) ) != EOF )
                                        putc( c, out_file );
                                putc( c, out_file);                  /* copy EOF */
                                printf("File has been copied.\n");
                                fclose( out_file);
                        }
                        fclose( in_file);
                }
        }
```

POINTERS TO FUNCTIONS
A pointer can also be declared as pointing to a function. The declaration of such a pointer is done by,

```
        int  (*func_pointer)();
```

The parentheses around **func_pointer* are necessary, else the compiler will treat the declaration as a declaration of a function. To assign the address of a function to the pointer, the statement,

```
        func_pointer = lookup;
```

where *lookup* is the function name, is sufficient. In the case where no arguments are passed to *lookup*, the call is

```
        (*func_pointer)();
```

The parentheses are needed to avoid an error. If the function *lookup* returned a value, the function call then becomes,

```
        i = (*func_pointer)();
```

If the function accepted arguments, the call then becomes,

```
        i = (*func_pointer)( argument1, argument2, argumentn);
```

SAMPLE CODE FOR POINTERS TO FUNCTIONS
Pointers to functions allow the creation of jump tables and dynamic routine selection. A pointer is assigned the start address of a function, thus, by typing the pointer name, program execution jumps to the routine pointed to.

By using a single pointer, many different routines could be executed, simply by re-directing the pointer to point to another function. Thus, programs could use this to send information to a printer, console device, tape unit etc, simply by pointing the pointer associated with output to the appropriate output function!

The following program illustrates the use of pointers to functions, in creating a simple shell program which can be used to specify the screen mode on a CGA system.

```
#include <stdio.h>      /* Funcptr.c */
#include <dos.h>

#define dim(x) (sizeof(x) / sizeof(x[0]) )
#define GETMODE        15
#define SETMODE         0
#define VIDCALL         0X10
#define SCREEN40        1
#define SCREEN80        3
#define SCREEN320       4
#define SCREEN640       6
#define VID_BIOS_CALL(x)  int86( VIDCALL, &x, &x )

int cls(), scr40(), scr80(), scr320(), scr640(), help(), shellquit();
union REGS regs;
```

```
struct command_table
{
  char *cmd_name;
  int (*cmd_ptr) ();
}
cmds[]={"40",scr40,"80",scr80,"320",scr320,"640",scr640,"HELP",help,"CLS",cls,"EXI
T",\
               shellquit};

cls()
{
  regs.h.ah = GETMODE;     VID_BIOS_CALL( regs );
  regs.h.ah = SETMODE;     VID_BIOS_CALL( regs );
}

scr40()
{
  regs.h.ah = SETMODE;
  regs.h.al = SCREEN40;
  VID_BIOS_CALL( regs );
}

scr80()
{
  regs.h.ah = SETMODE;
  regs.h.al = SCREEN80;
  VID_BIOS_CALL( regs );
}

scr320()
{
  regs.h.ah = SETMODE;
  regs.h.al = SCREEN320;
  VID_BIOS_CALL( regs );
}

scr640()
{
  regs.h.ah = SETMODE;
  regs.h.al = SCREEN640;
  VID_BIOS_CALL( regs );
}

shellquit()
{
   exit( 0 );
}

help()
{
   cls();
   printf("The available commands are; \n");
   printf("    40    Sets 40 column mode\n");
   printf("    80    Sets 80 column mode\n");
   printf("   320    Sets medium res graphics mode\n");
   printf("   640    Sets high res graphics mode\n");
   printf("   CLS    Clears the display screen\n");
   printf("  HELP    These messages\n");
   printf("  EXIT    Return to DOS\n");
}

get_command( buffer )
char *buffer;
{
  printf("\nShell: ");
  gets( buffer );
  strupr( buffer );
}

execute_command( cmd_string )
char *cmd_string;
{
  int i, j;
  for( i = 0; i < dim( cmds); i++ )
  {
    j = strcmp( cmds[i].cmd_name, cmd_string );
    if( j == 0 )
    {
      (*cmds[i].cmd_ptr) ();
      return 1;
    }
  }
  return 0;
}

main()
{
  char input_buffer[81];
  while( 1 )
  {
    get_command( input_buffer );
    if( execute_command( input_buffer ) == 0 )
      help();
  }
}
```

FORMATTERS FOR STRINGS/CHARACTERS
Consider the following program.

```
        #include <stdio.h>

        main()    /* FORMATS.C   */
        {
                char           c = '#';
                static char s[] = "helloandwelcometoclanguage";

                printf("Characters:\n");
                printf("%c\n", c);
```

```
            printf("%3c%3c\n", c, c);
            printf("%-3c%-3c\n", c, c);
            printf("Strings:\n");
            printf("%s\n", s);
            printf("%.5s\n", s);
            printf("%30s\n", s);
            printf("%20.5s\n", s);
            printf("%-20.5s\n", s);
    }
```

The output of the above program will be,

```
Characters:
#
  #  #
#  #
Strings:
helloandwelcometoclanguage
hello
    helloandwelcometoclanguage
               hello
hello
```

The statement printf("%.5s\n",s) means print the first five characters of the array s. The statement printf("%30s\n", s) means that the array s is printed right justified, with leading spaces, to a field width of thirty characters.

The statement printf("%20.5s\n", s) means that the first five characters are printed in a field size of twenty which is right justified and filled with leading spaces.

The final printf statement uses a left justified field of twenty characters, trailing spaces, and the .5 indicating to print the first five characters of the array s.

SYSTEM CALLS
Calls may be made to the Operating System to execute standard OPsys calls, eg,

```
    #include <process.h>
    main()  /* SYS.C    */
    {
            char *command = "dir";

            system( "cls" );
            system( command );
    }
```

Do not use this method to invoke other programs. Functions like exec() and spawn() are used for this.

Suggested Model Answers

Exercise C1 The program output is,

Prog1

```
        Programming in C is easy.
        And so is Pascal.
```

Prog2

```
        The black dog was big. The cow jumped over the moon.
```

Prog3

```
        Hello...
        ..oh my
        ...when do i stop?
```

Exercise C2 Typical program output is,

```
        The sum of 35 and 18 is 53
```

Exercise C3 Invalid variable names,

```
        value$sum       - must be an underscore, $ sign is illegal
        exit flag       - no spaces allowed
        3lotsofmoney    - must start with a-z or an underscore
        char            - reserved keyword
```

When %X\n is used, the hex digits a to f become A to F

Exercise C4 Constants

```
        #define smallvalue  0.312
        #define letter      'W'
        #define smallint    37
```

Exercise C5

```
        The % of 50 by 10 is 0.00
```

Exercise C6

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

```
main ()
{
        int  n = 1, t_number = 0;

        for ( ; n <= 200; n++ )
                t_number = t_number + n;

        printf("The 200th triangular number is %d\n", t_number);
}
```

Exercise C7

```
a == 2  this is an equality test
a  = 2  this is an assignment

/* program which illustrates relational assignments */
#include <stdio.h>

main()
{
        int val1 = 50, val2 = 20, sum = 0;

        printf("50 + 20 is %d\n", val1 + val2 );
        printf("50 - 20 is %d\n", val1 - val2 );
        printf("50 * 20 is %d\n", val1 * val2 );
        printf("50 / 20 is %d\n", val1 / val2 );
}
```

Exercise C8

```
Prints result with two leading places
```

Exercise C9

```
main()
{
        int  n = 1, t_number = 0, input;

        printf("Enter a number\n");
        scanf("%d", &input);
        for( ; n <= input; n++ )
                t_number = t_number + n;

        printf("The triangular_number of %d is %d\n", input, t_number);
}
```

Exercise C10

```
#include <stdio.h>

main()
```

```
{
        int grade;      /* to hold the entered grade */
        float average;  /* the average mark */
        int loop;       /* loop count */
        int sum;        /* running total of all entered grades */
        int valid_entry;        /* for validation of entered grade */
        int failures;   /* number of people with less than 65 */

        sum = 0;        /* initialise running total to 0 */
        failures = 0;

        for( loop = 0; loop < 5; loop = loop + 1 )
        {
                valid_entry = 0;
                while( valid_entry == 0 )
                {
                        printf("Enter mark (1-100):");
                        scanf(" %d", &grade );
                        if ((grade > 1 ) && (grade < 100 ))
                        {
                                valid_entry = 1;
                        }
                }
                if( grade < 65 )
                        failures++;
                sum = sum + grade;
        }
        average = (float) sum / loop;
        printf("The average mark was %.2f\n", average );
        printf("The number less than 65 was %d\n", failures );
}
```

Exercise C11

```
#include <stdio.h>

main ()
{
  int   invalid_operator = 0;
  char  operator;
  float number1, number2, result;

  printf("Enter two numbers and an operator in the format\n");
  printf(" number1  operator  number2\n");
  scanf( "%f %c %f", &number1, &operator, &number2);

  switch( operator )
  {
    case '*' : result = number1 * number2; break;
    case '-' : result = number1 - number2; break;
    case '/' : result = number1 / number2; break;
    case '+' : result = number1 + number2; break;
    default  : invalid_operator = 1;
  }
  switch ( invalid_operator )
  {
    case 1:  printf("Invalid operator.\n");     break;
    default: printf("%2.2f %c %2.2f is %2.2f\n",
             number1,operator,number2,result);  break;
  }
}
```

Exercise C12

```
max_value = 5
```

Exercise C13

```
#include <stdio.h>

main()
{
        static int m[][] = { {10,5,-3}, {9, 0, 0}, {32,20,1}, {0,0,8} };
        int row, column, sum;

        sum = 0;
        for( row = 0; row < 4; row++ )
                for( column = 0; column < 3; column++ )
                        sum = sum + m[row][column];
        printf("The total is %d\n", sum );
}
```

Exercise C14

Variables declared type static are initialised to zero. They are created and initialised only once, in their own data segment. As such, they are permanent, and still remain once the function terminates (but disappear when the program terminates).

Variables which are not declared as type static are type automatic by default. C creates these on the stack, thus they can assume non zero values when created, and also disappear once the function that creates them terminates.

Exercise C15

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

int calc_result( int var1, int var2, int var3 )
{
  int sum;

  sum = var1 + var2 + var3;
  return( sum );          /* return( var1 + var2 + var3 ); */
}

main()
{
  int numb1 = 2, numb2 = 3, numb3=4, answer=0;

  answer = calc_result( numb1, numb2, numb3 );
  printf("%d + %d + %d = %d\n", numb1, numb2, numb3, answer);
}
```

Exercise C16

```
#include <stdio.h>

int add2darray( int [][5], int );     /* function prototype */

int add2darray( int array[][5], int rows )
{
        int total = 0, columns, row;

        for( row = 0; row < rows; row++ )
                for( columns = 0; columns < 5; columns++ )
                        total = total + array[row][columns];
        return total;
}

main()
{
        int numbers[][] = { {1, 2, 35, 7, 10}, {6, 7, 4, 1, 0} };
        int sum;

        sum = add2darray( numbers, 2 );
        printf("the sum of numbers is %d\n", sum );
}
```

Exercise C17

```
time = time - 5;
a = a * (b + c);
```

Exercise C18

```
#include <stdio.h>
void sort_array( int [], int );

void sort_array( values, number_of_elements )
int values[], number_of_elements;
{
  int index_pointer, base_pointer = 0, temp;

  while ( base_pointer < (number_of_elements - 1) )
  {
    index_pointer = base_pointer + 1;
    while ( index_pointer < number_of_elements )
    {
      if( values[base_pointer] > values[index_pointer] )
      {
        temp = values[base_pointer];
        values[base_pointer]  = values[index_pointer];
        values[index_pointer] = temp;
      }
      ++index_pointer;
    }
    ++base_pointer;
  }
}

main ()
{
  static int array[] = { 4, 0, 8, 3, 2, 9, 6, 1, 7, 5 };
  int number_of_elements = 10, loop_count = 0;

  printf("Before the sort, the contents are\n");
  for ( ; loop_count < number_of_elements; ++loop_count )
    printf("Array[%d] is %d\n", loop_count,array[loop_count]);

  sort_array( array, number_of_elements );

  printf("After the sort, the contents are\n");
  loop_count = 0;
  for( ; loop_count < number_of_elements; ++loop_count )
    printf("Array[%d] is %d\n", loop_count,array[loop_count]);
}
```

Exercise C19

```
#include <stdio.h>
long int triang_rec( long int );

long int triang_rec( long int number )
{
    long int result;

    if( number == 0l )
      result = 0l;
    else
      result = number + triang_rec( number - 1 );
    return( result );
}

main ()
{
  int request;
  long int triang_rec(), answer;

  printf("Enter number to be calculated.\n");
  scanf( "%d", &request);

  answer = triang_rec( (long int) request );
  printf("The triangular answer is %l\n", answer);
}
```

```
Note this version of function triang_rec

#include <stdio.h>
long int triang_rec( long int );

long int triang_rec( long int number )
{
   return((number == 0l) ? 0l : number*triang_rec( number-1));
}
```

Exercise C20

b

Exercise C21

```
#include <stdio.h>

struct date {
        int day, month, year;
};

int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
struct date today, tommorrow;

void gettodaysdate( void );

void gettodaysdate( void )
{
        int valid = 0;

        while( valid == 0 ) {
                printf("Enter in the current year (1990-1999)-->");
                scanf("&d", &today.year);
                if( (today.year < 1990) || (today.year > 1999) )
                        printf("\007Invalid year\n");
                else
                        valid = 1;
        }
        valid = 0;
        while( valid == 0 ) {
                printf("Enter in the current month (1-12)-->");
                scanf("&d", &today.month);
                if( (today.month < 1) || (today.month > 12) )
                        printf("\007Invalid month\n");
                else
                        valid = 1;
        }
        valid = 0;
        while( valid == 0 ) {
                printf("Enter in the current day (1-%d)-->",
days[today.month-1]);
```

```
                scanf("&d", &today.day);
                if( (today.day < 1) || (today.day > days[today.month-1]) )
                        printf("\007Invalid day\n");
                else
                        valid = 1;
        }
}

main()
{

        gettodaysdate();
        tommorrow = today;
        tommorrow.day++;
        if( tommorrow.day > days[tommorrow.month-1] ) {
                tommorrow.day = 1;
                tommorrow.month++;
                if( tommorrow.month > 12 )
                        tommorrow.year++;
        }
        printf("Tommorrows date is %02d:%02d:%02d\n", \
                tommorrow.day, tommorrow.month, tommorrow.year );
}
```

Exercise C22

```
#include <stdio.h>

struct  date  {         /* Global definition of date */
        int day, month, year;
};

main()
{
        struct date dates[5];
        int i;

        for( i = 0; i < 5; ++i ) {
                printf("Please enter the date (dd:mm:yy)" );
                scanf("%d:%d:%d", &dates[i].day, &dates[i].month,
                        &dates[i].year );
        }
}
```

Exercise C23

```
count = 10, x = 10;

Q Q
/ /
( (
```

Exercise C24

```
i1 = 5, i2 = 12, *p1 = 5; *p2 = 5
```

Exercise C25

```
        Name = Baked Beans
        ID = 312
        Price = 2.75
```

Exercise C26

```
        Name = Apple Pie
        ID = 123
        Price = 1.65

        Name = Greggs Coffee
        ID = 773
        Price = 3.20
```

Exercise C27

```
        #include <stdio.h>

        main( int argc, char *argv[])
        {
          FILE *in_file, *out_file, *fopen();
          int c;

          if( argc != 3 )
          {
            printf("Incorrect, format is FCOPY source dest\n");
            exit(2);
          }
          in_file = fopen( argv[1], "r");
          if( in_file == NULL )  printf("Cannot open %s for reading\n", argv[1]);
          else
          {
            out_file = fopen( argv[2], "w");
            if ( out_file == NULL )   printf("Cannot open %s for writing\n",
argv[2]);
            else
            {
               printf("File copy program, copying %s to %s\n", argv[1],  argv[2]);
               while ( (c=getc( in_file) ) != EOF )  putc( c, out_file );
               putc( c, out_file);                    /* copy EOF */
               printf("File has been copied.\n");
               fclose( out_file);
            }
          fclose( in_file);
          }
        }
```

Practise Exercise 1: Answers

```
1.      int  sum;

2.      char  letter;

3.      #define  TRUE  1

4.      float  money;

5.      double  arctan;

6.      int  total = 0;

7.      int loop;

8.      #define GST  0.125
```

Practise Exercise 2: Answers

```
1.      total = number1;

2.      sum = loop_count  +  petrol_cost;

3.      discount = total  /  10;

4.      letter = 'W';

5.      costing = (float) sum  / 0.25;
```

Practise Exercise 3: Answers

```
1.      printf("%d", sum );

2.      printf("Welcome\n");

3.      printf("%c", letter );

4.      printf("%f", discount );

5.      printf("%.2f", dump );

6.      scanf("%d", &sum );

7.      scanf("%f", &discount_rate );

8.      scanf("  %c", &operator );
```

Practise Exercise 4: Answers

```
1.      for(  loop = 1;  loop <= 10; loop++ )
            printf("%d\n", loop );

2.      for(  loop = 1; loop <= 5; loop++ ) {
           for( count = 1; count <= loop; count++ )
```

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

3.        total = 0;
          for(  loop = 10; loop <= 100; loop++ )
             total = total + loop;

          or

          for(  loop = 10, total = 0; loop <= 100; loop++ )
             total = total + loop;

5.        for( loop = 'A';  loop <= 'Z';  loop++ )
             printf("%c", loop );
```

Practise Exercise 5: Answers

```
1.        loop = 1;
          while( loop <= 10 ) {
                  printf("%d", loop );
                  loop++;
          }

2.        loop = 1;
          while ( loop <= 5 ) {
                  count = 1;
                  while( count <= loop )
                          printf("%d", loop);
                  printf("\n");
          }

3.        if( sum < 65 )
                  printf("Sorry. Try again");

4.        if( total == good_guess )
                  printf("%d", total );
          else
                  printf("%d", good_guess );
```

Practise Exercise 6: Answers

```
1.        if( (sum == 10)  && (total < 20) )
                  printf("incorrect.");

2.        if(  (flag == 1)  ||  (letter != 'X') )
                  exit_flag = 0;
          else
                  exit_flag = 1;

3.        switch( letter ) {
                  case 'X' : sum = 0; break;
                  case 'Z' : valid_flag = 1; break;
                  case 'A' : sum = 1; break;
```

```
                default: printf("Unknown letter -->%c\n", letter ); break;
        }
```

Practise Exercise 7: Answers

```
1.      char  letters[10];

2.      letters[3] = 'Z';

3.      total = 0;
        for( loop = 0; loop < 5; loop++ )
                total = total + numbers[loop];

4.      float  balances[3][5];

5.      total = 0.0;
        for( row = 0; row < 3; row++ )
                for( column = 0; column < 5; column++ )
                        total = total + balances[row][column];

6.      char  words[] = "Hello";

7.      strcpy(  stuff, "Welcome" );

8.      printf("%d", totals[2] );

9.      printf("%s", words );

10.     scanf(" %s", &words[0] );

        or

        scanf("  %s", words );

11.     for( loop = 0; loop < 5; loop++ )
           scanf(" %c", &words[loop] );
```

Practise Exercise 8: Answers

```
1.      void  menu( void )
        {
                printf("Menu choices");
        }

2.      void  menu( void );

3.      void  print(  char  message[] )
        {
                printf("%s", message );
        }

4.      void  print(  char  [] );

5.      int  total( int  array[], int elements )
```

```
        {
                int  count, total = 0;
                for( count = 0; count < elements; count++ )
                        total = total + array[count];
                return total;
        }
```

```
6.      int  total(  int  [],  int );
```

Practise Exercise 9: Answers

```
1.      struct  client {
                int     count;
                char  text[10];
                float  balance;
        };

2.      struct  date  today;

3.      struct  client  clients[10];

4.      clients[2].count = 10;

5.      printf("%s", clients[0].text );

6.      struct birthdays
        {
                struct time  btime;
                struct date  bdate;
        };
```

Practise Exercise 9A: Answers

```
1.      FILE *input_file;

2.      input_file = fopen( "results.dat", "rt" );

3.      if( input_file == NULL ) {
                printf("Unable to open file.\n");\
                exit(1);
        }

4.      int ch, loop = 0;

        ch = fgetc( input_file );
        while( ch != '\n' ) {
                buffer[loop] = ch;
                loop++;
                ch = fgetc( input_file );
        }
        buffer[loop] = NULL;

5.      fclose( input_file );
```

Practise Exercise 10: Answers

```
1.      int  *address;

2.      temp = &balance;

3.      *letter = 'W';

4.      count = 20, *temp = 20, sum = 20

5.      char  *message = "Hello";

6.      array = (char *) getmem( 200 );
```

Practise Exercise 11: Answers

```
1.      struct  date  *dates;

2.      (*dates).day = 10;
        or
        dates->day = 10;

3.      struct  machine  {
                int  name;
                char  *memory;
        };

4.      mpu641->memory = (char *) NULL;

5.      mpu641->memory = CPUtype;

        [       -> means mpu641 is a pointer to a structure                  ]
        [       memory is a pointer, so is assigned an address (note &)
]
        [       the name of an array is equivalent to address of first element
]

6.      mpu641->name = 10;

        [       -> means mpu641 is a pointer to a structure          ]
        [       name is a variable, so normal assignment is possible ]


7.      *(times->day) = 10;

        [       -> means times is a pointer to a structure          ]
        [       day is a pointer, so to assign a value requires * operator   ]
        [       *times->day is not quite correct                             ]
        [       using the pointer times, goto the day field          ]       times-
>day
        [       this is an address                                   ]       x
        [       let the contents of this address be equal to 10             ]
*(x) = 10
```

```
8.      *(times[2]->month) = 12;
```

Practise Exercise 11a: Answers

```
1. Before call to editrecord()
        item.name = "Red Plum Jam"
        item.id = 0
        item.price = 0.0

2. After return from editrecord()
        item.name = "Baked Beans"
        item.id = 220
        item.price = 2.20

3. The final values of values, item.name, item.id, item.price
        item.name = "Baked Beans"
        item.id = 220
        item.price = 2.75
```

Practise Exercise 12: Answers

```
1.      struct  node  {
                int  data;
                struct  node  *next_node;
        };

2.      struct  node  node1, node2, node3;

3.      node1.next = &node2;
        node2.next = &node3;
        node3.next = (struct node *) NULL;

4.      while( list != (struct node *) NULL ) {
                printf("data = %d\n", list->data );
                list = list->next_node;
        }

5.      terminates the list at node2, effectively deleting node3 from the list.

6.      new_node->next = list->next;
        list->next = new_node;

7.      void  delete_node( struct node *head, struct node *delnode )
        {
                struct node *list;

                list = head;
                while( list->next != delnode ) {
                        list = list->node;

                list->next = delnode->next;
        }
```

```
8.      void insert_node( struct node *head, struct node *newnode, struct node
*prevnode )
        {
                struct node *list;

                list = head;
                while( list != prevnode )
                        list = list->next;

                newnode->next = list->next;
                list->next = newnode;
        }
```

Practise Exercise 13: Answers

```
1.      FILE *input_file;

2.      input_file = fopen( "results.dat", "rt" );

3.      if( input_file == NULL ) {
                printf("Unable to open file\n");
                exit( 1 );
        }

4.      loop = 0;
        ch = fgetc( input_file );
        while( (ch != '\n') && (ch != EOF) ) {
                buffer[loop] = ch;
                loop++;
                ch = fgetc( input_file );
        }
        buffer[loop] = 0;

5.      fclose( input_file );
```