Chapter 1

An Overview of C

1.0. Quick History of C

- Developed at Bell Laboratories in the early seventies by Dennis Ritchie.

- Born out of two other languages – BCPL(Basic Control Programming Language) and B.

- C introduced such things as character types, floating point arithmetic, structures, unions and the preprocessor.

- The principal objective was to devise a language that was easy enough to understand to be "high-level" – i.e. understood by general programmers, but low-level enough to be applicable to the writing of systems-level software.

- The language should abstract the details of how the computer achieves its tasks in such a way as to ensure that C could be portable across different types of computers, thus allowing the UNIX operating system to be compiled on other computers with a minimum of re-writing.

- C as a language was in use by 1973, although extra functionality, such as new types, was introduced up until 1980.

- In 1978, Brian Kernighan and Dennis M. Ritchie wrote the seminal work *The C Programming Language,* which is now the standard reference book for C.

- A formal ANSI standard for C was produced in 1989.

- In 1986, a descendant of C, called C++ was developed by Bjarne Stroustrup, which is in wide use today. Many modern languages such as C#, Java and Perl are based on C and C++.

- Using C language scientific, business and system-level applications can be developed easily.

1.1. The Compile Process

You type in your program as C source code, compile it (or try to run it from your programming editor, which will typically compile and build automatically) to turn it into object code, and then build the program using a linker program to create an executable program that contains instructions written in machine code, which can be understood and run on the computer that you are working on.

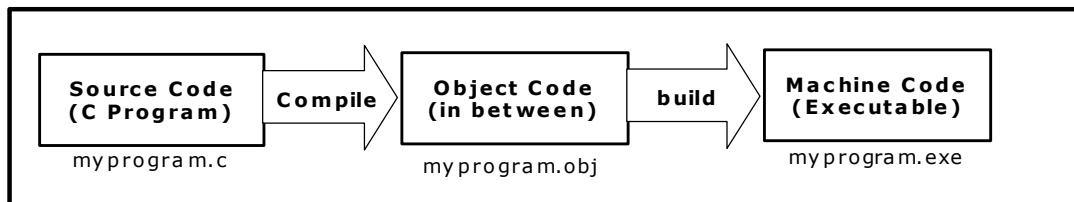| Source Code (C Program)	Compile	Object Code (in between)	build	Machine Code (Executable)
myprogram.c		myprogram.obj		myprogram.exe

Fig 1.0. Compilation Process

To compile a C program that you have typed in using C/C++ Editor, do one of the following:

- Click on the menu and choose **Compile**.
- Hold down the **Alt** key and press **F9**

To run the program, do one of the following:

- Click on the menu and choose **Run**.
- Press Ctrl and F9 together.
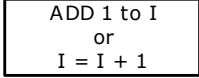
1.2. Guidelines on Structure:
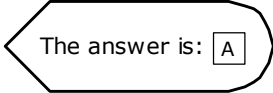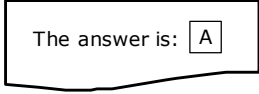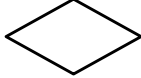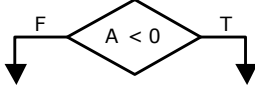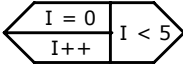
When creating your programs, here are a few guidelines for getting a program that can be easily read and debugged by you (when you come back to it later) and others (who may have to change your program later).
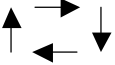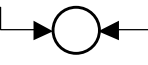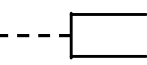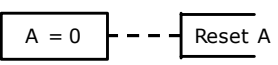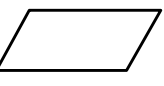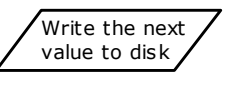
- A well-structured program should indent (i.e. move to the right by pressing the tab key once) all instructions inside each block (i.e. following an opening curly brace – {). Therefore, a block inside another block will be indented twice etc. This helps to show which instructions will be executed, and also helps to line up opening and closing curly braces.

- It is often helpful to separate stages of a program by putting an extra blank line (and maybe a comment explaining what the next stage does) between each stage.

- Another useful technique that helps the readability inside each instruction is to put spaces after commas when listing parameters (i.e. data) to be given to a function. This helps identify where each parameter starts and makes the program easier to read. Spaces before and after brackets, assignments (=) and operators (+ - * / < > == != etc.) can also help make things more readable – but this is up to you.

- Start a program with a **/* comment */** that explains the purpose of the program, and who wrote it. This is useful in case you ever have to work on somebody else's program, and you need to ask them for advice on how it works.

- If you create a new function, put a comment before it showing what data goes in, and what comes out, and the purpose of the function.

- If a part of your code is quite difficult to follow, or you wish to explain why you have written it the way you have, or even if you wish to say that it could be improved (and how) but you haven't the time to do it better yet, then use a comment.

1.3. FLOWCHARTING:

Note: In the examples below, assume that variables **A** and **I** are **integers**

Symbol	Type of operation/C Code Example	Example of symbol usage
	Terminal Activity - Start, Stop or End {	START

	Assignment of a value to a variable, either directly or as the result of a calculation. $I = I + 1;$	ADD 1 to I or I = I + 1
	Softcopy - screen output to a video display. printf ("The answer is: %d", A);	The answer is: A
	Hardcopy - document output from a printer.	The answer is: A

	Decision based on a condition.
	
	if (A < 0)
	{
	    statements;
	}
	else
	{
	    statements;
	}
		F A < 0 T

	To repeat a statement/s a known number of times.
	
	for (I = 0; I < 5; I++)
	{
	    statements;
	}
		I = 0
		I++ I < 5

	Sub-routine (function) used to indicate a process, which is defined elsewhere. INTRO (); /* Call Intro*/	INTRO
	Connectors: On-page (left) & Off-page (right). Used to either: 1. Continue a flowchart at a different place either on or off the same piece of paper. 2. Close a selection branch or loop.	L R

	Flow of Control Arrows indicating the sequence of steps ("flow of control").	
	Annotation for placing comments in logic. A = 0; /* Reset A */	

	General Input/Output of Data
	
	fprintf (filename, data);
	printf("Message");
	scanf("%d", &I);	

1.3.1.   General Flowcharting Guidelines

Symbols can be of any size (height or width) necessary to hold the symbol's contents. The shapes of most symbols imply the process. It is redundant to put the word "print" in a hardcopy symbol for example.

Always put an arrowhead on each line connecting symbols to indicate the "flow of control". The only symbols that may receive more than one incoming arrow are connectors. Never enter any other symbols using more than one arrow. If you want to do that, put a connector in front of the symbol and let the multiple arrows enter the connector.

Don't do this:	Instead, do this:
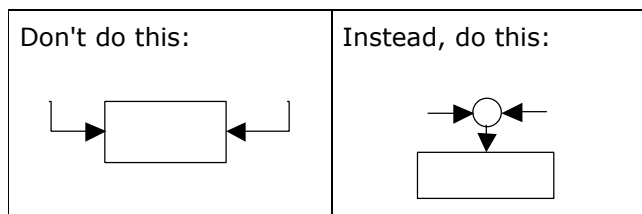
The switch statement involves a special use of the diamond symbol. A flowchart for the following switch statement is shown to its right. Notice that the diamond contains only the name of the single variable to be evaluated. The "legs" that exit the decision diamond are each labeled with the unique values from a limited set of possible values for the variable answer, including the "else" (default) option.

switch (ANS)
{
        case 'Y':
        case 'y':
            X = 1;
            break;
        case 'N':
        case 'n':
            X =-1;
            break;
        default:
            X = 0;
 	
}	
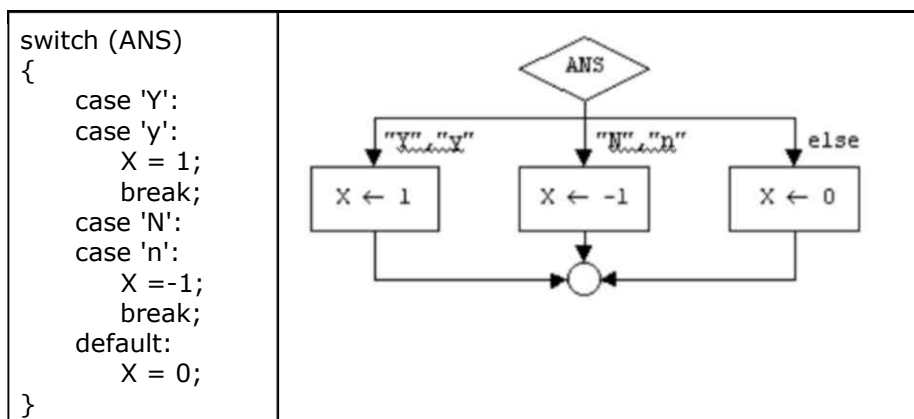
1.4.   Algorithm

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

The ordered set of instructions required to solve a problem is known as an *algorithm*.

1.5.   Using the Turbo C/C++ Editor:

Most of the commands can be activated directly without going through the main menu. These operations are activated by using hot keys and can be used whenever needed.

Hot Key	Meaning
F1	On line help
F2	Save
F3	Select a file to load
F4	Execute program until cursor is reached
F5	Zoom window
F6	Switches between windows
F7	Traces into function calls
F8	Trace but skip function calls
F9	Compile and link program
F10	Toggle between main menu and editor
Alt F1	Shows the previous help screen
Alt F3	Close an open file
Alt F6	Switch between watch and message windows
Alt F7	Previous error
Alt F8	Next error
Alt F9	Compiles file to .obj
Alt spacebar	Activates the main menu
Alt C	Compile menu
Alt D	Debug menu
Alt E	Editor
Alt F	File menu
Alt O	Options menu
Alt P	Project menu
Alt R	Run menu
Alt X	Quit TC

Ctrl F1	C Help about the item upon which the cursor is present
Ctrl F2	Reset program
Ctrl F3	Show function call stack
Ctrl F4	Evaluate an expression
Ctrl F7	Set a watch expression
Ctrl F8	Set or clear a break point
Ctrl F9	Execute program

1.5.1.   Moving, Deleting and Copying Blocks of Text:

Start of block	CTRL – KB	
End of block	CTRL – KK	
Move block	CTRL – KV	Removes previous defined block and places it at the new location.
Copy block	CTRL – KC	
Print a block	CTRL – KP	
Delete block mark	CTRL - KY	

1.5.2.   Deleting Characters, Words and Lines:

To delete entire word to the right of cursor: CTRL-T

Entire line: CTRL-Y

Current cursor position to the end of line: CTRL-QY

1.5.3.   Cursor Movement:

HOME	Move cursor to the start of the line
END	Cursor to the end of the line
CTRL-PGUP	Cursor to the top of the file
CTRL-PGDN	Cursor to the bottom of the file
CTRL-HOME	Cursor to the top of the screen
CTRL-END	Cursor to the bottom of the screen
CTRL - ←	Left one word
CTRL - →	Right one word

1.5.4.   Moving Blocks of Text to and from Disk Files:

1.   Define the Block:

Start of block: CTRL – KB

End of Block: CTRL – KK

2.   Then type: CTRL - KW

It asks for file name to move the block

3.   To Read a block in: CTRL – KR

You are prompted for the file name and the contents come to the current cursor location.

1.6.   Information Representation:

The two common standards for representing characters:

1.   ASCII (American standard code for information interchange).

2.   EBCDIC (Extended Binary coded decimal interchange code)

Character	ASCII	EBCDIC
A	65	193
1	49	241
+	43	78

C allows a character constant to be specified as a character enclosed in single quote marks (' ') which the compiler will then translate to a value in the code (ASCII or otherwise).

1.7.   Characteristics of C:

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

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

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

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

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

1.8. Compiling & Executing C Program:


```
                        ┌──────────┐
                        │  Editor  │
                        └──────────┘
        ┌──────────────┐  ┌────────────┐  ┌──────────────┐
        │ Source Code  │─▶│Preprocessor│◀─│ Header Files │
        └──────────────┘  └────────────┘  └──────────────┘
                               │
                          ┌──────────┐
                          │ Compiler │
                          └──────────┘
                               │
                          ┌──────────┐
                          │Assembler │
                          └──────────┘
                               │
                        ┌──────────────┐
                        │ Object Code  │
                        └──────────────┘
                               │
        ┌──────────────┐  ┌──────────┐  ┌──────────────┐
        │ Object Files │─▶│  Linker  │◀─│  Libraries   │
        └──────────────┘  └──────────┘  └──────────────┘
                               │
                        ┌────────────────┐
                        │Executable files│
                        └────────────────┘
```
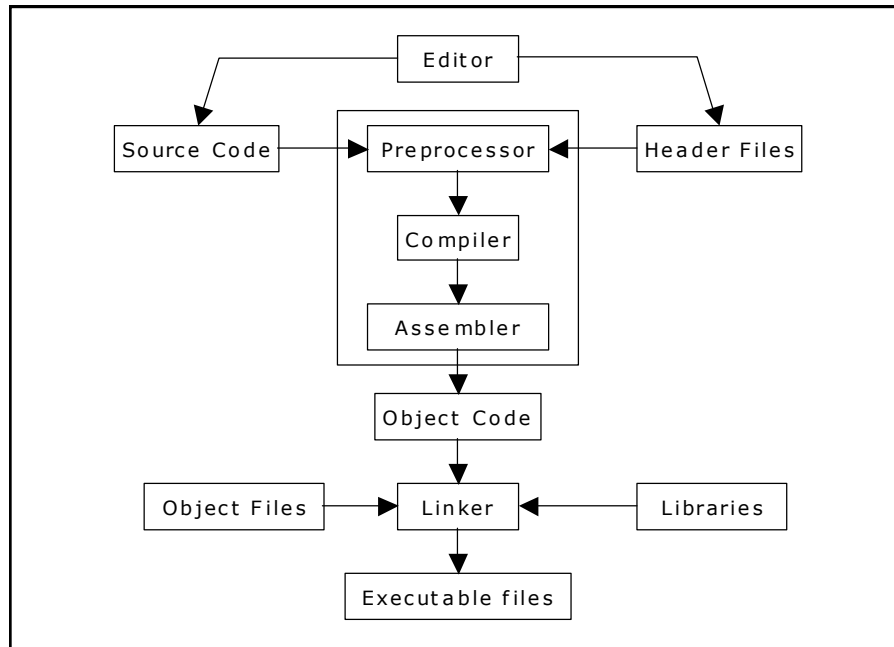
Stages of Compilation and Execution

C language program is passed through the following steps:

1. C preprocessor: This program accepts C source files as input and produces a file that is passed to the compiler. The preprocessor is responsible for removing comments and interpreting special C language preprocessor directives, which can be easily identified as they begin with the # symbol. These special preprocessor statements libraries include the following.

    a) # include: The preprocessor adds the information about the object code used in the body of the main function. These files are called header files.

    b) # define: this directive assigns a symbolic name to a constant. Symbolic names are used to make programs more readable and maintainable.

    c) # ifdef: This directive is called conditional compilation and allows the programmer to write programs that are adaptable to different operating environment.

2. C compiler: This program translates the C language source code into the machine assembly language.

3. Assembler: The assembler accepts the C – compiler output and creates object code. If the program does not contain any external function calls, this code is directly executable.

4. Linker: If a source file references library functions, which is defined in other source files, the linker combines these functions with the main() function to create an executable program file.

1.9. Keywords in C:

Here are the 32 keywords:

Flow control (6) – if, else, return, switch, case, default

Loops (5) – for, do, while, break, continue

Common *types* (5) – int, float, double, char, void

For dealing with *structures* (3) – struct, typedef, union

Counting and sizing things (2) – enum, sizeof

Rare but still useful *types* (7) – extern, signed, unsigned, long, short, static, const

Keywords that is undiscouraged and which we NEVER use (1) – goto

We don't use unless we're doing something strange (3) – auto, register, volatile

Total keywords: **32**

1.10. Simple C program structure:

In C, comments begin with the sequence /* and are terminated by */. Any thing that is between the beginning and ending comments symbols is ignored by the compiler.

Example:
> /* sample program */

In C, blank lines are permitted and have no effect on the program. A non-trivial C application typically has following general portions on it:

The structure of a C program looks as follows:

```
Source code:

    Header Files:
          # includes

    Manifest constants:
          # defines

    User supplied function prototypes

    Global variable definitions

    int main (void)
    {
          Local variable definitions
          -- body of the program --
    }

    User written functions
```
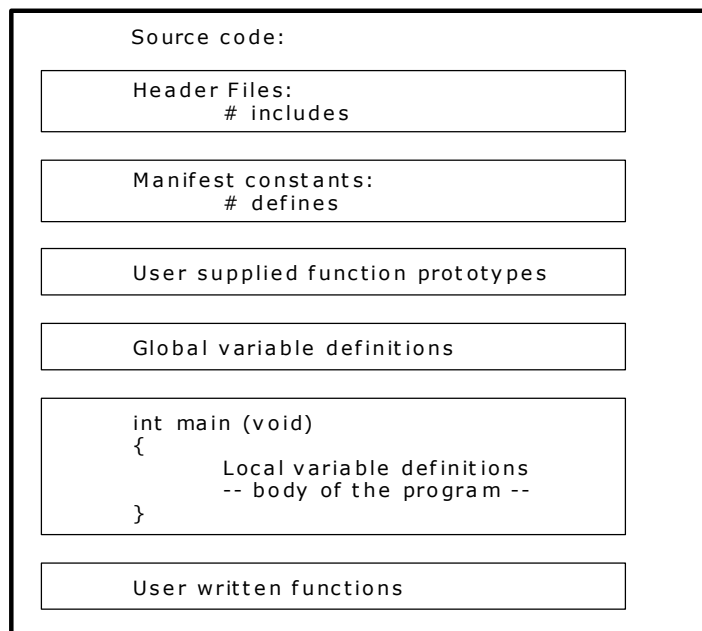
Header Files (.h):

Header files contains declaration information for function or constants that are referred in programs. They are used to keep source-file size to a minimum and to reduce the amount of redundant information that must be coded.

 # includes:

An include directive tells the preprocessor to include the contents of the specified file at the point in the program. Path names must either be enclosed by double quotes or angle brackets.

Example:

1:      # include <stdio.h>

2:      # include "mylib.h"

3:      # include "mine\include\mylib.h"

In the example (1) above, the <> tells the preprocessor to search for the included file in a special known \include directory or directories.

In the example (2), the double quotes (" ") indicate that the current directory should be checked for the header file first. If it is not found, the special directory (or directories) should be checked.

The example (3) is similar, but the named relative directory \mine\include is checked for the header file mylib.h.

Relative paths can also be proceeded by the .\ or ..\ notation; absolute paths always begin with a \.

 # defines:

ANSI C allows you to declare **constants.** The # define directive is used to tell the preprocessor to perform a search-and-replace operation.
Example:

        # define Pi 3.14159

        # define Tax-rate 0.0735

In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the # define line.

User supplied function prototypes:

Declares the user-written functions actually defined later in the source file. A function prototype is a statement (rather than an entire function declaration) that is placed a head of a calling function in the source code to define the function's label before it is used. A function prototype is simply a reproduction of a function header (the first line of a function declaration) that is terminated with a semi-colon.

Global variable definitions:

Create the Global variables before main ().

The main function:

main ()   The main function is the entry point in the C program. All C programs begin execution by calling the main () function. When the main function returns, your program terminates execution and control passes back to the operating system.

Every C/C++ program must have one and only one main function.

{   The next line consists of a single curly brace which signifies the start of main () function.

int age ;   The first line of code inside function main () is declaration of variables. In C all variables must be declared before they are used.

}   which signifies the end of main () function.

User-written functions:

Divide the application into logical procedural units and factor out commonly used code to eliminate repetition.

1.11. Fundamental/Primary Data Types:

C has the following simple data types (16-bit implementation):

Type	Size	Range	Precision for real numbers
char	1 byte	-128 to 127	
unsigned char	1 byte	0 to 255	
signed char	1 byte	-128 to 127	
short int or short	2 bytes	-32,768 to 32,767	
unsigned short or unsigned short int	2 bytes	0 to 65535	
int	2 bytes	-32,768 to 32,767	
unsigned int	2 bytes	0 to 65535	
Long or long int	4 bytes	-2147483648 to 2147483647 (2.1 billion)	
unsigned long or unsigned long int	4 bytes	0 to 4294967295	
float	4 bytes	3.4 E–38 to 3.4 E+38	6 digits of precision
double	8 bytes	1.7 E-308 to 1.7 E+308	15 digits of precision
long double	10 bytes	+3.4 E-4932 to 1.1 E+4932	provides between 16 and 30 decimal places

The syntax of the simple type specifier:
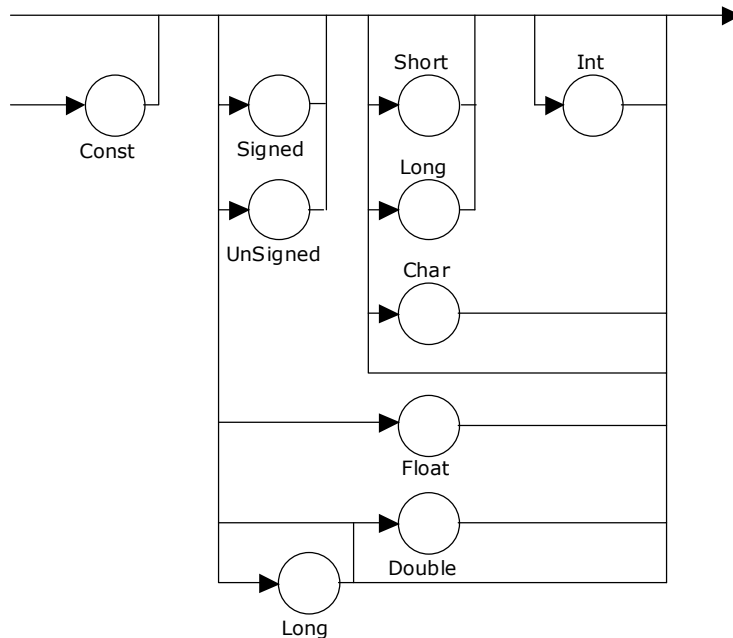
Currently the three char data types are guaranteed to be 1 byte in length, but the other data types are machine architecture dependent. Unsigned can be used with all char and int types.

1.12. Identifier names (Declaring Variables):

The C language defines the names that are used to reference variables, functions, labels, and various other user-defined objects as identifiers.

An identifier can vary from one to several characters. The first character must be a letter or an underscore with subsequent characters being letters, numbers or under-score.

Example:
        Correct: Count, test23, High_balances, _name
        In-correct: 1 count, hil there, high..balance

In turbo C, the first 32 characters of an identifier name are significant. If two variables have first 32 characters in common and differ only on the 33rd, TC will not be able to tell then apart. In C++ identifiers may be any length.

In C, upper and lower case characters are treated as different and distinct.

Example:
        count, Count, COUNT are three separate identifiers.

To declare a variable in C, do:

        var_type **list variables**;

 Example:
        int i, j, k;
        float x, y, z;
        char ch;
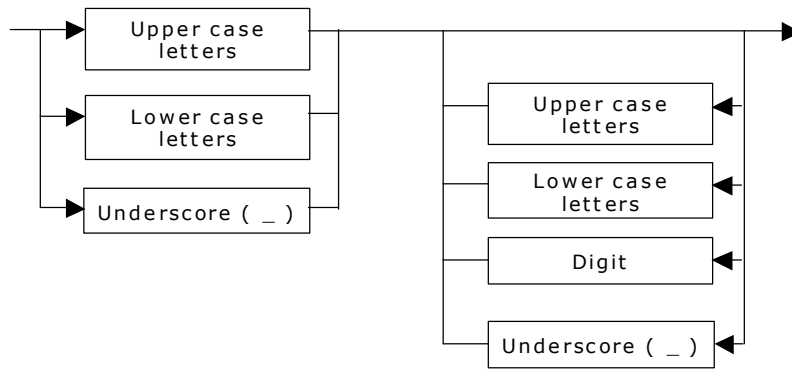
1.13. Declaring Variables:

Variables are declared in three basic places: inside functions, in the definition of function parameters and outside of the functions. If they are declared inside functions, they are called as *local variables*, if in the definition of functions then formal parameters and out side of all functions, then global variables.

1.13.1.   Global variables:

Global variables are known throughout the program. The variables hold their values throughout the programs execution. Global variables are created by declaring them outside of any function.

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

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

main()
{

}
```

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

Example:

```
float sum = 0.0;
int bigsum = 0;
char letter =`A';
main()
{

}
```

This is the same as:

```
float sum;
int bigsum;
char letter;
main()
{
```

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

is more efficient.

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

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

which is the same as, but more efficient than:

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

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

1.13.2.   Local variables:

Variable that are declared inside a function are called "local variables". These variables are referred to as "automatic variables". Local variables may be referenced only by statements that are inside the block in which the variables are declared.

Local variables exist only while the block of code in which they are declared is executing, i.e. a local variable is created upon entry into its block & destroyed upon exit.

Example:

Consider the following two functions:

```
    void func1 (void)
    {
            int x;
            x = 10;
    }

    void func2 (void)
    {
            int x;
            x = -199;
    }
```

The integer variable x is declared twice, once in func1 () & once in func2 (). The x in func1 () has no bearing on or relationship to the x in func2 (). This is because each x is only known to the code with in the same block as variable declaration.

1.14. Manifest Constants:

The # define directive is used to tell the preprocessor to perform a search-and-replace operation.

Example: # define Pi 3.14159
                # define Tax-rate 0.0735

In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the # define line.

The following are two purposes for defining and using manifest constants:

      (1)   They improve source-code readability
      (2)   They facilitate program maintenance.

1.15. Constants:

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

Syntax for initializing a const data type:

     Const data-type variable = initial value.

Example:

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

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

The difference between a const variable and a manifest constant is that the # define causes the preprocessor to do a search-and—replace operation through out code. This sprinkles the literal through your code wherever it is used. On the other hand, a const variable allows the compiler to optimize its use. This makes your code run faster (compiler optimization is outside the scope of this course).

1.16. Escape sequences:

C provides special backslash character constants as shown below:

Code	Meaning
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\'	Single quote
\"	Double quote
\\	Back slash
\a	Alert (bell)
\f	Form feed

1.17. Operators:

There are three general classes of operators: arithmetic, relational and logical and bitwise.

1.17.1.   Arithmetic Operators:

Operation	Symbol	Meaning
Add	+	
Subtract	-	
Multiply	*	
Division	/	Remainder lost
Modulus	%	Gives the remainder on integer division, so 7 % 3 is 1.
--	Decrement	
++	Increment	

As well as the standard arithmetic operators (+ - * /) found in most languages, C provides some more operators.

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

Increment ++, Decrement -- which are more efficient than their long hand equivalents.

For example, X = X + 1 can be written as ++X or as X++. There is however a difference when they are used in expression.

The ++ and -- operators can be either in post-fixed or pre-fixed. A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1. Consider the following:

        X = 10;
        Y = ++X;

In this case, Y will be set to 11 because X is first incremented and then assigned to Y. However if the code had been written as

        X = 10;
        Y = X++;

Y would have been set to 10 and then X incremented. In both the cases, X is set to 11; the difference is when it happens.

The % (modulus) operator only works with integers.

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

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

RULE: If both arguments of / are integer then do integer division. So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 / 2.0.

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

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

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

We can rewrite $i = i + 3$ as $i += 3$

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

NOTE: that $x *= y + 2$ means $x = x * (y + 2)$ and <u>NOT</u> $x = x * y + 2$.

1.17.2.   Relational Operators:

The relational operators are used to determine the relationship of one quantity to another. They always return 1 or 0 depending upon the outcome of the test. The relational operators are as follows:

Operator	Action
= =	equal to
! =	not equal to
<	Less than
<=	less than or equal to
>	Greater than
>=	Greater than or equal to

To test for equality is ==

If the values of x and y, are 1 and 2 respectively then the various expressions and their results are:

Expression	Result	Value
X != 2	False	0
X == 2	False	0
X == 1	True	1
Y != 3	True	1

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

        if (i = j) .....

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

Not equals is: !=

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

1.17.3. Logical (Comparison) Operators:

Logical operators are usually used with conditional statements. The three basic logical operators are && for logical AND, || for logical OR and ! for not.

The truth table for the logical operators is shown here using one's and zero's. (the idea of true and false under lies the concepts of relational and logical operators). In C true is any value other than zero, false is zero. Expressions that use relational or logical operators return zero for false and one for true.

P	Q	P && q	P || q	! p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Example:

(i)   x == 6 && y == 7

This while expression will be TRUE (1) if both x equals 6 and y equals 7, and FALSE (0) otherwise.

(ii)   x < 5 || x > 8

This whole expression will be TRUE (1) if either x is less than 5 or x is greater than 8 and FALSE (0) other wise.

1.17.4. Bit wise Operators:

The **bit wise** operators of C are summarised in the following table:

Bitwise operators	
&	AND
|	OR
^	XOR
~	One's Compliment
<<	Left shift
>>	Right Shift

The truth table for Bitwise operators AND, OR, and XOR is shown below. The table uses 1 for true and 0 for false.

P	Q	P AND q	P OR q	P XOR q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

DO NOT confuse & with &&: & is bit wise AND, && logical AND. Similarly for | and ||.

~ is a unary operator: it only operates on one argument to right of the operator. It finds 1's compliment (unary). It translates all the 1 bits into O's and all O's into 1's

Example:

        12 = 00001100
        ~12 = 11110011 = 243

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator <u>must</u> be positive. The vacated bits are filled with zero (***i.e.*** when shift operation takes places any bits shifted off are lost).
Example:

        X << 2 shifts the bits in X by 2 places to the left.
So:

        if X = 00000010 (binary) or 2 (decimal)
then:
X >>= 2 implies X = 00000000 or 0 (decimal)

Also: if X = 00000010 (binary) or 2 (decimal)

X <<= 2 implies X = 00001000 or 8 (decimal)

Therefore a shift left is equivalent to a multiplication by 2.

Similarly, a shift right is equal to division by 2.

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 ***use shifts***.

The bit wise AND operator (&) returns 1 if both the operands are one, otherwise it returns zero. For example, if y = 29 and z = 83, x = y & z the result is:

        0 0 0 1 1 1 0 1   29 in binary
                                            &
        0 1 0 1 0 0 1 1   83 in binary

        0 0 0 1 0 0 0 1   Result

The bit wise or operator (|) returns 1 if one or more bits have a value of 1, otherwise it returns zero. For example if, y = 29 and z = 83, x = y | z the result is:

        0 0 0 1 1 1 0 1   29 in binary
                                            |
        0 1 0 1 0 0 1 1   83 in binary

        0 1 0 1 1 1 1 1   Result

The bit wise XOR operator (^) returns 1 if one of the operand is 1 and the other is zero, otherwise if returns zero. For example, if y = 29 and z = 83, x = y ^ z the result is:

        0 0 0 1 1 1 0 1   29 in binary
                                         ^
        0 1 0 1 0 0 1 1   83 in binary

        0 1 0 0 1 1 1 0   Result

1.17.5.   Conditional Operator:

Conditional expression use the operator symbols question mark (?)

(x > 7) ? 2 : 3

What this says is that if x is greater than 7 then the expression value is 2. Otherwise the expression value is 3.

In general, the format of a conditional expression is:
        a ? b : c

Where, a, b & c can be any C expressions.

Evaluation of this expression begins with the evaluation of the sub-expression 'a'. If the value of 'a' is true then the while condition expression evaluates to the value of the sub-expression 'b'. If the value of 'a' is FALSE then the conditional expression returns the value of the sub-expression 'C'.

1.18. sizeof Operator:

In situation where you need to incorporate the size of some object into an expression and also for the code to be portable across different machines the size of unary operator will be useful. The size of operator computes the size of any object at compile time. This can be used for dynamic memory allocation.

        Usage: sizeof (object)
The object itself can be the name of any sort of variable or the name of a basic type (like int, float, char etc).

Example:
            sizeof (char) = 1
            sizeof (int) = 2
            sizeof (float) = 4
            sizeof (double) = 8

1.19. Special Operators:

Some of the special operators used in C are listed below. These are reffered as separators or punctuators.

Ampersand (&)   Comma (,)   Asterick (*)
Ellipsis (…)   Braces ({ })   Hash (#)
Brackets ([ ])   Parenthesis (())   Colon (:)
Semicolon (;)

Ampersand:

Ampersand (&) also referred as address of operator usually precedes the identifier name, which indicates the memory allocation (address) of the identifier.

Comma:

Comma (,) operator is used to link the related expressions together. Comma used expressions are linked from left to right and the value of the right most expression is the value of the combined expression. The comma operator has the lowest precedence of all operators. For example:

Sum = (x = 12, y = 8, x + y);

The result will be sum = 20.

The comma operator is also used to separate variables during declaration. For example:

int a, b, c;

Asterick:

Asterick (*) also referred as an indirection operator usually precedes the identifier name, which identifies the creation of the pointer operator. It is also used as an unary operator.

Ellipsis:

Ellipsis (…) are three successive periods with no white space in between them. It is used in function prototypes to indicate that this function can have any number of arguments with varying types. For example:

void fun (char c, int n, float f, ...)

The above declaration indicates that fun () is a function that takes at least three arguments, a char, an int and a float in the order specified, but can have any number of additional arguments of any type.

Hash:

Hash (#) also referred as pound sign is used to indicate preprocessor directives, which is discussed in detail already.

Parenthesis:

Parenthesis () also referred as function call operator is used to indicate the opening and closing of function prototypes, function calls, function parameters, etc., Parenthesis are also used to group expressions, and there by changing the order of evaluation of expressions.

Semicolon:

Semicolon (;) is a statement terminator. It is used to end a C statement. All valid C statements must end with a semicolon, which the C compiler interprets as the end of the statement. For example:

c = a + b;
b = 5;

1.20. Order of Precedence of C Operators:

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

We may want the effect as either
(a + b) * c
or
a + (b * c)

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

    a - b - c

is evaluated as: (a - b) - c

as you would expect.

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

Highest	( ) [ ] -> .
	! ~ - (type) * & sizeof ++ --
	* / %
	+ -
	<< >>
	< <= >= >
	== !=
	&
	^
	|
	&&
	||
	?:
	= += -= *= /= etc.
Lowest	, (comma)

Thus: a < 10 && 2 * b < c is interpreted as: (a < 10) && ((2 * b) < c)

1.21. Pair Matching:

Turbo C++ editor will find the companion delimiters for the following pairs:

    { }, [ ], ( ), < >, /* */, " " and ` `.

To find the matching delimiter, place the cursor on the delimiter you wish to match and press CTRL – Q [for a forward match and CTRL – Q] for a reverse match. The editor will move the cursor to the matching delimiter.

1.22. Casting between types:

Mixing *types* can cause problems. For example:

```
int a = 3;
int b = 2;
float c;
c = b * (a / b);
printf ("2 * (3 / 2) = %f\n", c);
```

doesn't behave as you might expect. Because the first (a/b) is performed with integer arithmetic it gets the answer 1 not 1.5. Therefore the program prints:

    2 * (3/2) = 2.000

The best way round this is what is known as a *cast*. We can *cast* a variable of one type to another type like so:

```
int a = 3;
int b = 2;
float c;
c = b * ( (float) a / b);
```

The (float) a construct tells the compiler to switch the type of variable to be a float. The value of the expression is automatically cast to float .   The main use of *casting* is when you have written a routine which takes a variable of one type and you want to call it with a variable of another type. For example, say we have written a power function with a prototype like so:

```
int pow (int n, int m);
```

We might well have a float that we want to find an approximate power of a given number *n*. Your compiler should complain bitterly about your writing:

```
float n= 4.0;
int squared;
squared= pow (n, 2);
```

The compiler will not like this because it expects *n* to be of type int but not float.

However, in this case, we want to tell the compiler that we do know what we're doing and have good reason for passing it a float when it expects an int (whatever that reason might be). Again, a cast can rescue us:

```
float n = 4.0;
int squared;
squared = pow ((int) n, 2);      /* We cast the float down to an int*/
```

IMPORTANT RULE: To move a variable from one type to another then we use a *cast* which has the form (*variable_type*) *variable_name*.

CAUTION: It can be a problem when we *downcast* – that is cast to a type which has less precision than the type we are casting from. For example, if we cast a double to a float we will lose some bits of precision. If we cast an int to a char it is likely to overflow [recall that a char is basically an int which fits into 8 binary bits].

1.23. Console I/O:

Console I/O refers to operations that occur at the keyboard and screen of your computer.

1.23.1.   Reading and writing Characters:

The simplest of the console I/O functions are getche (), which reads a character from the keyboard, and putchar (), which prints a character to the screen. The getche () function waits until a key is pressed and then returns its value. The key pressed is also echoed to the screen automatically. The putchar () function will write its character argument to the screen at the current cursor position. The prototypes for getche () and putchar () are shown here:

```
int getche (void);
int putchar (int c);
```

The header file for getche () and putchar () is in CONIO.H.

The following programs inputs characters from the keyboard and prints them in reverse case. That is, uppercase prints as lowercase, the lowercase prints as uppercase. The program halts when a period is typed. The header file CTYPE.H is required by the islower() library function, which returns true if its argument is lowercase and false if it is not.

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

main(void)
{
        char ch;
        printf ("enter chars, enter a period to stop\n");
        do
        {
                ch = getche ();
                if ( islower (ch) )
                        putchar (toupper (ch));
                else
                        putchar (tolower (ch));
        } while (ch! = '.');                    /* use a period to stop */
        return 0;
}
```

There are two important variations on getche().

- The first is getchar(), which is the original, UNIX-based character input function.

    ❑ The trouble with getchar() is that it buffers input until a carriage return is entered. The reason for this is that the original UNIX systems line-buffered terminal input, i.e., you had to hit a carriage return for anything you had just typed to actually be sent to the computer.

    ❑ The getchar() function uses the STDIO.H header file.

- A second, more useful, variation on getche() is getch(), which operates precisely like getche () except that the character you type is not echoed to the screen. It uses the CONIO.H header.

1.23.2.   Reading and writing Strings:

On the next step, the functions gets() and puts() enable us to read and write strings of characters at the console.

The gets() function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. We can type characters at the keyboard until we strike a carriage return. The carriage return does not become part of the string; instead a null terminator is placed at the end and gets() returns. Typing mistakes can be corrected prior to striking ENTER. The prototype for gets() is:

        char* gets (char *str);

Where, str is a character array that receives the characters input by the user. Its prototype is found in STDIO.H. The following program reads a string into the array str and prints its length.

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

main(void)
{
        char str[80];
        gets (str);
        printf ("length is %d", strlen(str));
        return 0;
}
```

The puts() function writes its string argument to the screen followed by a newline. Its prototype is.

    int puts (char str);

It recognizes the same backslash codes as printf(), such as "\t" for tab. It cannot output numbers or do format conversions. Therefore, puts() takes up less space and runs faster than printf(). Hence, the puts() function is often used when it is important to have highly optimized code. The puts() function returns a non negative value if successful, EOF otherwise. The following statement writes "hello" on the screen.

    puts ("hello");

The puts() function uses the STDIO.H header file.

Basic console I/O functions:

Function	Operation
getchar()	Reads a character from the keyboard and waits for carriage return
getche()	Reads a character with echo and does not waits for carriage return
getch()	Reads a character from the keyboard with out echo and not waits for carriage return
Putchar()	Writes a character to the screen
gets()	Reads a string from the keyboard
puts()	Writes a string to the screen

Distinguishion between getchar() and gets() functions:

getchar()	gets()
Used to receive a single character.	Used to receive a single string, white spaces and blanks.
Does not require any argument.	It requires a single argument.

1.23.3.   Reading Character Data in a C Program

All data that is entered into a C program by using the scanf function enters the computer through a special storage area in memory called the **standard input buffer** (or **stdin**). A user's keystrokes (including the new line character \n generated when the Enter key is pressed) are stored in this buffer, which can then be read using functions such as scanf. When numbers are read from this area, the function converts the keystrokes stored in

the buffer (except for the \n) into the type of data specified by the control string (such as "%f") and stores it in the memory location indicated by the second parameter in the scanf function call (the variable's address). The \n remains in the buffer. This can cause a problem if the next scanf statement is intended to read a *character* from the buffer. The program will mistakenly read the remaining \n as the character pressed by the user and then proceed to the next statement, never even allowing the user to enter a character of their own.

You can solve this problem when writing C statements to read *character* data from the keyboard by adding a call to a special function named **fflush** that clears all characters (including \n's) from the given input buffer. The statement would be place ahead of each statement in your program used to input characters, such as:

    fflush(stdin); scanf("%c", &A);
            or
    fflush(stdin); A=getchar();

1.24. Formatted Console I/O (printf() and scanf()):

The pre-written function printf() is used to output most types of data to the screen and to other devices such as disks. The C statement to display the word "Hello" would be:

    printf("Hello");

The printf () function can be found in the stdio.h header file.

1.24.1.   Displaying Prompts:

The printf() function can be used to display "prompts" (messages urging a user to enter some data) as in:

    printf ("How old are you? ");

When executed, this statement would leave the cursor on the same line as the prompt and allow the user to enter a response following it on the same line. A space was included at the end of the prompt (before the closing quote mark) to separate the upcoming response from the prompt and make the prompt easier to read during data entry.

1.24.2.   Carriage Returns:

If you want a carriage return to occur after displaying a message, simply include the special *escape sequence* \n at the end of the of the message before the terminating quote mark, as in the statement.

    printf ("This is my program.\n");

1.24.3.   Conversion Specifiers:

All data output by the printf() function is actually produced as a string of characters (one symbol after another). This is because display screens are character-based devices. In order to send any other type of data (such as integers and floats) to the screen, you must add special symbols called conversion specifiers to the output command to convert data from its stored data format into a string of characters. The C statement to display the floating point number 123.456 would be:

    printf ("%f",123.456);

The "%f" is a conversion specifier. It tells the printf() function to convert the floating point data into a string of characters so that it can be sent to the screen. When outputting non-string data with the printf() function you must include two parameters (items of data) within the parentheses following printf(). The first parameter is a quoted control string containing the appropriate conversion specifier for the type of non-string data being displayed *and optionally* any text that should be displayed with it. The second parameter (following a comma) should be the value or variable containing the value. The C statement to display the floating point number 123.456 preceded by the string "The answer is: " and followed by a carriage return would be:

printf ("The answer is: %f\n",123.456);

Notice that the value does not get typed inside of the quoted control string, but rather as a separate item following it and separated by a comma. The conversion specifier acts as a place holder for the data within the output string.

The following table lists the most common conversion specifiers and the types of data that they convert:

Specifier	Data Type
%c	char
%f	float
%d or %i	signed int (decimal)
%h	short int
%p	Pointer (Address)
%s	String of Characters
Qualified Data Types	
%lf	long float or double
%o	unsigned int (octal)
%u	unsigned int (decimal)
%x	unsigned int (hexadecimal)
%X	Unsigned Hexadecimal (Upper Case)
%e	Scientific Notation (Lower case e)
%E	Scientific Notation (Upper case E)
%g	Uses %e or %f which ever is shorter
%ho	short unsigned int (octal)
%hu	short unsigned int (decimal)
%hx	short unsigned int (hexadecimal)
%lo	long unsigned int (octal)
%lu	long unsigned int (decimal)
%lx	long unsigned int (hexadecimal)
%Lf	long double
%n	The associated argument is an integer pointer into which the number of characters written so far is placed.
%%	Prints a % sign

1.24.4.   Output Field Width and Rounding:

When displaying the contents of a variable, we seldom know what the value will be. And yet, we can still control the format of the output field (area), including the:

amount of space provided for output (referred to as the output's "*field width*")
alignment of output (left or right) within a specified field and rounding *of floating point numbers* to a fixed number of places right of the decimal point

Output formatting is used to define specific alignment and rounding of output, and can be performed in the printf() function by including information within the conversion specifier. For example, to display the floating point number 123.456 right-aligned in an eight character wide output field and rounded to two decimal places, you would expand basic conversion specifier from "%f" to "%8.2f". The revised statement would read:

    printf ("The answer is:%8.2f\n",123.456);

The 8 would indicate the width, and the .2 would indicating the rounding. *Keep in mind that the addition of these specifiers as no effect on the value itself, only on the appearance of the output characters.*

The value 123.456 in the statement above could be replaced by a variable or a symbolic constant of the same data type. The 8 in the statement above specifies the width of the output field. If you include a width in the conversion specifier, the function will attempt to display the number in that width, aligning it against the rightmost character position. Unused positions will appear as blank spaces (padding) on the left of the output field. If you want the value to be left-aligned within the field, precede the width with a minus sign (-). If no width is specified, the number will be displayed using only as many characters as are necessary without padding. When a values is too large to fit in a specified field width, the function will expand the field to use as many characters as necessary.

The .2 in the statement above specifies the decimal precision (i.e., the number of place that you want to appear to the right of the decimal point) *and would be used only in situations where you are outputting floating point values*. The function will round the output to use the specified number of places (adding zeros to the end if necessary). Any specified field width *includes* the decimal point and the digits to its right. The default decimal precision (if none is specified) is 6 places.

In situations where you want floating point values displayed in scientific notation, formatting also is used to define specific alignment and rounding of output in the printf function by including information within the conversion specifier %e. For example, to display the floating point number 123.456 in scientific notation in a twelve character wide output field with its mantissa (significant digits) rounded to two decimal places, you would expand basic conversion specifier from "%e" to "%12.2e". The resulting output would be:

    The answer is: 1.23e+002

Notice that values displayed in scientific notation always place the decimal point after the first significant digit and use the exponent (digits shown following the letter e) to express the power of the number. The C statement to produce the output above would be:

    printf ("The answer is:%12.2e\n",123.456);

The 12 would indicate the *overall* field width (following any message) *including* the decimal point and the exponential information. The .2 would indicating the rounding of the digits following the decimal point. The exponential information is always expressed in

5 characters: the first one an "e", then a sign (- or +), followed by the power in three characters (with leading zeros if needed).

Review the following examples of formatted output statements paying close attention to the format of the resulting output beside them. Each box indicates one character position on the screen. All output starts in the leftmost box, although some output might be "padded" with blank spaces to align it to the right edge of the field. "X"'s indicated unused positions.

C command using printf () with various conversion specifiers:	Output Produced on Screen																			
	Position:									1	1	1	1	1	1	1	1	1	2
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
printf ("%3c",'A');			A	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-3c",'A');	A			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%8s","ABCD");					A	B	C	D	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%d",52);	5	2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%8d",52);							5	2	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-8d",52);	5	2							X	X	X	X	X	X	X	X	X	X	X	X
printf ("%f",123.456);	1	2	3	.	4	5	6	0	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10f",123.456);	1	2	3	.	4	5	6	0	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10.2f",123.456);					1	2	3	.	4	6	X	X	X	X	X	X	X	X	X	X
printf ("%-10.2f",123.456);	1	2	3	.	4	6					X	X	X	X	X	X	X	X	X	X
printf ("%.2f",123.456);	1	2	3	.	4	6	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%10.3f",-45.8);				-	4	5	.	8	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10f",0.00085);			0	.	0	0	0	8	5	0	X	X	X	X	X	X	X	X	X	X
printf ("%10.2e",123.89);			1	.	2	4	e	+	0	0	2	X	X	X	X	X	X	X	X	X

Distinguishing between printf() and puts() functions:

puts()	printf()
They can display only one string at a time.	They can display any number of characters, integers or strings a time.
All data types of considered as characters.	Each data type is considered separately depending upon the conversion specifications.

1.23. scanf () function:

scanf() is the general purpose console input routine. It can read all the built-in data types of automatically convert numbers into the proper internal format. It is much like the reverse of printf(). The f in scanf stands for formatted.

```
# include <stdio.h>
int main()
{
        int num;
        printf ("Enter a number: ");
        scanf("%d", &num);
        printf("The number you have entered was %d\n", num);
        return 0;
}
```

The above program requires a variable in which to store the value for num. The declaration, int num; provides a temporary storage area called num that has a data type of integer (whole number). The scanf function requires a Format String, which is provided by the %d character pair. The percent sign is an introducer and is followed by a conversion character, d, which specifies that the input is to be an integer. The input is stored in the variable, num. The scanf function requires the address of the variable, and this is achieved by prefixing the variable name with an ampersand (eg. &num).

The printf statement also uses the format string to relay the information. The printf function does not require the address of the variable and so the ampersand is not required. The prototype for scanf() is in STDIO.H.

The format specifiers for scanf () are as follows:

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer, hexa decimal or octal integer
%h	Read a short integer
%e	Read a floating-point number
%f	Read a floating-point number
%g	Read a floating-point number
%o	Read an octal number
%s	Read a string
%x	Read a hexadecimal number
%p	Read a pointer
%n	Receives an integer value equal to the number of characters read so far
%u	Read an unsigned integer
%[..]	Scan for a string of words

Distinguishing between scanf() and gets() functions:

scanf()	gets()
Strings with spaces cannot be accessed until ENTER key is pressed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character data type can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers.	Only one string can be received at a time. Strings, floats can be received at a time.

Key Terms & Concepts

The list of terms below is provided to supplement or elaborate on the boldfaced terms and definitions provided in the course textbook. Students are advised to also review the textbook to develop a fuller understanding of these and other important terms related.

Text is a form of data consisting of characters such as letters, numerals, and punctuation.

ASCII is the American Standard Code for Information Interchange; a language for representing text on computers.

Binary is a word with two meanings in programming. The first and most common meaning relates to the numbering system known as "Base-2" in which all values are represented using only "two (bi) numerals (nary) ". The second meaning relates to the use of operators such as the minus sign (-) in a formula. When the symbol appears between *two* items (such as the values 5 and 2 in the formula 5-2), it is referred to as a "*binary* operator". When the symbol appears preceding only *one* item (such as the value 2 in the formula -2), it is referred to as a "*unary* operator".

Bits are binary digits (0 or 1) that represent the settings of individual switches or circuits inside a computer (OFF or ON). Data is represented as standardized patterns of bits.

A **byte** is the unit of measure of storage space used to represent a single character. There are typically 8 bits in a byte.

Decimal is another word with two meanings in programming. The first and most common meaning relates to the numbering system known as "Base-10" in which all values are represented using ten digits (0-9). The second meaning relates to the use of a decimal point when writing numbers with fractional parts, such as one half in 8.5 or one tenth in 4.1. Numbers written containing a decimal point are often referred to as "decimal numbers", however this is technically incorrect, as it implies that the numbers are also written using the Base-10 numbering system (which may not be the case). A more precise way to talk about numbers that contain fractional parts is to call them "floating point numbers" or more simply "floats".

Bug: A bug is an error in a program. These can be split into syntax errors and semantic errors. A Syntax error is where the language rules of the C programming language have been broken, and can be easily picked up by the C Compiler – e.g. a missing semi-colon. A semantic error is harder-to find, and refers to an error where the syntax of the program is correct, but the meaning of the program is not what was intended – e.g. int three_cubed=3*3; when you actually meant int three_cubed=3*3*3;

Breakpoint: This is a line of source code, marked in some way, where the program execution stops temporarily for you to examine what is happening – e.g. the contents of variables etc. Once stopped, you can usually step through the execution of the program line-by-line to see how variables change, and to follow the flow of execution in the program.

Build: This is the step following compilation. It takes one or a number of object code files generated by one or more compiled programs, and links them together to form an executable file that comprises instructions that are understood by the computer on which the application is to run.

Execution: This means running a program – i.e. starting the executable file going so that we can see the program working.

Debugging: Debugging refers to the task of removing bugs from a program. This is often attained with the help of tools in the IDE, such as creating breakpoints, stepping through line-by-line etc.

Executable: The name given to a file on your computer that contains instructions written in machine code. These are typically applications that do some task on your computer – e.g. *Microsoft Word* has an executable file (winword.exe) that you can execute in order to start the application.

IDE: Integrated Development Environment. The application used to edit your source code, perform debugging tasks, compile and build your application, and organize your projects. It acts as a complete set of tools in which to create your C programs.

Program: This is a generic name for an application on your computer that performs a task. An application such as *Microsoft Word* may be referred to as a program. Sometimes also used to refer rather vaguely as the source code that, when compiled, will generate the executable application.

Run: When we refer to running a program, we are referring to the execution (starting) of the executable version of the program so that we can see the program working.

Machine Code: The language that each type of computer understands. All PCs understand instructions based on microprocessors such as the Intel 8086 through to the latest Pentium processors, Macintoshes understand instructions based on the 68000 and upwards processors, and Sun servers understand instructions based on Sun's Sparc processor.

Syntax: The syntax refers to the structure of a C program. It refers to the rules that determine what is a correct C program. A Syntax Error is where a part of the program breaks these rules in some way – e.g. a semi-colon omitted from the end of a C instruction.

A **compiler** is a program that translates high-level language (such as C) instructions into machine language instructions. It also often provides an editor for writing the program code in the first place. Compilers often are packaged with other software known as programming environments that include features for testing and debugging programs.

Source Code: The C Program that you type in using a text editor, containing a list of C instructions that describe the tasks you wish to perform. The source code is typically fairly independent of the computer that you write it on – the compile and build stages convert it to a form that is understood by your computer.

Object code is machine language that resulted from a compiling operation being performed. Files containing C object code typically end with a filename extension of ".obj".

Text Editor: An application program, rather like a word-processor, that you use to type out your C source code instructions to create a C program. These can be as simple as the notepad text editor supplied with Microsoft Windows, to the sophisticated editor that comes part of an IDE, typically including features such as bookmarks, integrated debugging, and syntax highlighting.

A **header file** is a file containing useful blocks of pre-written C source code that can be added to your source code using the "# include" compiler directive. Header files typically end with a filename extension of ".h".

White space is any character or group of characters that a program normally interprets as a separator of text, including: spaces ( ), form feeds (\f), new-lines (\n), carriage returns (\r), horizontal tabs (\t), and vertical tabs (\v). In C source code, all of these characters are interpreted as the same unless they are quoted. In other words, one space is interpreted the same as three blank lines.

The **declaration** part of a program defines its identifiers, such as: symbolic constants and variable names and data types.

The **body of a program** contains the statements that represent the steps in the main algorithm.

A **constant** (also called as literal) is an actual piece of data (also often referred to as a "value") such as the number 5 or the character 'A'.

Symbolic **Constants** are aliases (nicknames) used when coding programs in place of values that are expected to be the same during each execution of a program, but might need to be changed someday. Symbolic constants are defined within C source code using the "#define" compiler directive at the top of the program to allow easy location and revision later.

Variable: A temporary storage location in a C program, used to hold data for calculations (or other uses) further in on in the program. For identification purposes, a variable is given a name, and to ensure we hold the right sort of information in the variable, we give it a type (e.g. int for whole numbers, float for decimal numbers). An example would be **int count, total;** or **char name[21];** or **float wage_cost, total_pay;**

A **keyword** is a reserved word within a program that can NOT be redefined by the programmer.

Identifiers are labels used to represent such items such as: constants, variables, and functions. Identifiers are case-sensitive in C, meaning that upper and lowercase appearances of the same name are treated as being different identifiers. Identifiers that are defined within a function (see below) are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined outside *of* all functions are global, meaning that they will be recognized within all functions.

Addresses are numeric designations (like the numbers on mailboxes) that distinguish one data storage location from another. Prior to the use of identifiers, programmers had to remember the address of stored data rather than its identifier. In C, addresses are referred to by preceding an identifier with an ampersand symbol (&) as in &X which refers to the address of storage location X as opposed to its contents.

Integer data is a numeric type of data involving a whole number (i.e. it CANNOT HAVE a fractional portion). An example of an integer constant is 5 (written without a decimal point). The most common type identifier (reserved word used in a declaration) for integer data is int.

Floating point data is a numeric type of data that CAN HAVE a fractional portion. Mathematicians call such data a real number. An example of a floating point **constant is** 12.567 (written without a decimal point). The most common type identifiers for floating point data are float (for typical numbers) and double (for numbers such as 1.2345678901234E+205 that involve extreme precision or magnitude).

Character data is a type of data that involves only a single symbol such as: 'A', '4', '!', or ' ' (a blank space). The type identifier for character data is char.

String data is a type of data involving multiple symbols such as words or sentences. The C language stores strings as a collection of separate *characters* (see above).

Boolean data is a logical type of data involving only two values: True or False. The identifier used in C to declare Boolean data is bool.

A **problem statement** is program documentation that defines the purpose and restrictions of a program in sufficient detail that the program can be analyzed and designed without more facts.

A sample **softcopy** is program documentation that precisely demonstrates all <u>video</u> (or other intangible) output required of a program.

A sample **hardcopy** is program documentation that precisely demonstrates all <u>printed</u> (tangible) output required of a program.

A **structure diagram** is graphic documentation that helps to describe the hierarchical relationship between a module and its various sub-modules.

An **algorithm** is a finite list of steps to follow in solving a well-defined task, written in the language of the user (i.e. in human terms).

Pseudocode is an algorithm written in English, but as clearly stated logical items that can be easily translated by a programmer into C or other high-level programming languages.

A **desk check** is a manual test of a program algorithm that is performed <u>prior</u> to writing the source code. It must follow the algorithm <u>exactly</u> and typically produces two items of documentation: the tracing chart showing what values are stored in memory during program execution, and any test outputs (softcopy and hardcopy) showing that the algorithm will produce the output specified earlier in the samples.

A **logic error** is caused by a mistake in the steps used to design the program algorithm.

A **syntax error** is caused by a grammatical error in the language of the source code.

A **run-time error** occurs when a program encounters commands it cannot execute.

Comments can be included within C source code, enclosed in the symbols /* and */. The inclusion of comments in source code neither causes the program to run more slowly, nor causes the object code to take up more space, since comments are not translated.

Blank spaces in C code act as separators and are not allowed in an identifier. Multiple blank spaces are treated the same as one, except in string constants (text inside of double quotes).

Character constants must be enclosed in single quote marks (').

String constants must be enclosed in double quote marks (").

Escape sequences are special strings typed within C output statements to produce characters that would otherwise be interpreted as having special meaning to the compiler.

Semicolon (**;**) Each statement is C is normally terminated with a semi-colon (;) except for: include statements.

A **compiler directive** (or preprocessor directive) is an instruction in a C source code file that is used to give commands to the compiler about how the compilation should be performed (as distinguished from C language statements that will be translated into machine code). Compiler directives are not *statements*, therefore they are not terminated with semi-colons. Compiler directives are written starting with a # symbol immediately in front of (touching) the command word, such as #include <stdio.h>

include is a compiler directive in C that is used to indicate that a unit of pre-defined program code (such as the header file stdio.h) should be linked to your program when it is compiled.

define is a compiler directive in C that is used to indicate that a symbolic constant is being used in your program in place of a constant value and that the value should be used in place of the symbolic constant when translating the source code into machine language.

{ (open brace) is the symbol used in C to start a group of executable statements.

} (close brace) is the symbol used in C to end a group of executable statements.

printf is the name of a function in C that will display formatted output on the screen.

scanf is the name of a function in C that will store input from a keyboard into a variable and then advance the cursor to the next line when the user presses the Enter key.

Assignment is the action of storing a value in a memory location and is accomplished using the symbol =.

An **expression** is another name for a formula. Expressions consist of operands (such as constants or variables) and operators (such as + or -). Operators that involve data from *only one* operand are called unary operators. In the statement X = -5; the minus sign acts as a unary operator and operates on a single operand (the constant 5). Operators that involve data from *two* operands are called binary operators. In the statement X = A/B; the slash acts as a binary operator and operates on a pair of operands (the variables A and B).

An **arithmetic expression** is one which uses numeric operands (such as constants or variables) and mathematical operations (such as addition, subtraction, multiplication, or division) to produce a numeric result. An example of an arithmetic expression is X+3.

Order of Precedence is a term used to describe which operations precede others when groups of operators appear in an expression. For example, C compilers see the expression A+B/C as A+(B/C) as opposed to (A+B)/C. The division operation will preceed the addition because division has a higher order of precedence.

Casting is the process of converting data from one data type to another. For example, if you try to divide two integer variables A and B and store the result in a floating point variable C, the result will have any decimal fraction truncated (chopped-off) because both operands are integers. To prevent this, *cast* either of the operands as floats *before* the division operation is performed, as in either of the following examples:

        C = (float) A / B;
        C = A / (float) B;
Trying to cast the result instead of the operands would be pointless because the truncation would have already taken place. So it would be ineffective to try:

        C = (float) (A / B);

Top-Down Design is an analysis method in which a major task is sub-divided into smaller, more manageable, tasks called functions (see definition below). Each sub-task is then treated as a completely new analysis project. These sub-tasks may be sufficiently large and complex that they also require sub-division, and so on. The document that analysts use to represent their thinking related to this activity is referred to as a structure diagram. For more information and an example of such a diagram, see the web notes on Analysis & coding of a task involving multiple functions.

Boolean data is a logical type of data with only two values: True and False. Boolean data can be represented in two ways in C. When you want to store Boolean data for fututre use, a variable can be declared to have data type of _Bool (notice the leading underscore and capitalization). A common alternative approach is to represent the true and false values using the integer values of 1 for true and 0 for false. Many functions in C use the latter approach and return a 1 or a 0 to indicate if a condition is true or false. In C, *any* non-zero integer value is interpreted as true.

Ordinal data is a type of data in which all of the values within the set are known and in a predictable order. Ordinal data types include: all integer data types and char data, but not floating point or string data.

Relational operators are those used to evaluate the relationships between items of data, including:

  == *for Equal to, the opposite of which is written in C as != (or not equal to).*
  > - Greater than, the opposite of which is written in C as <= (less than or equal to).
  < - Less than, the opposite of which is written in C as >= (greater than or equal to).

Relational expressions (also known as relational tests) describe conditions using formulae such as X==A+B that compare items of data (possibly including constants, symbolic constants, variables or arithmetic expressions) and produce a boolean (true or false) result.

Logical operators such as && for "and", || for "or" and ! for "not" are those used to combine conditions into logical expressions such as (X==0 && Y>10) to produce a single boolean result.

A **condition** is any expression that produces a boolean result.

Branching is the act of breaking out of the normal sequence of steps in an algorithm to allow an alternative process to be performed or to allow the repetition of process(es).

Chapter 2

Control Statements, Arrays and Strings

2.0.   Control Statements:

This deals with the various methods that C can control the *flow* of logic in a program. Control statements can be classified as un-conditional and conditional branch statements and loop or iterative statements. The Branch type includes:

1. Un-conditional:

- goto
- break
- return
- continue

2. Conditional:

- if
- if – else
- Nested if
- switch case statement

3. Loop or iterative:

- for loop
- while loop
- do-while loop

2.1.   Conditional Statements:

Sometimes we want a program to select an action from two or more alternatives. This requires a deviation from the basic sequential order of statement execution. Such programs must contain two or more statements that might *be* executed, but have some way to select only one of the listed options each time the program is run. This is known as conditional execution.

2.1.1.   if statement:

Statement or set of statements can be conditionally executed using if statement. Here, logical condition is tested which, may either true or false. If the logical test is true (non zero value) the statement that immediately follows if is executed. If the logical condition is false the control transfers to the next executable statement.

The general syntax of simple **if** statement is:

if (*condition*)
  statement_to_execute_if_condition_is_true;

 or

if (*condition*)
{
  statement 1;
  statement 2;
   _ _ _ _;
}

Flowchart Segment:
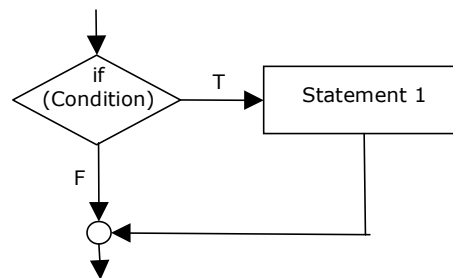
2.1.2. if – else statement:

The if statement is used to execute only one action. If there are two statements to be executed alternatively, then if-else statement is used. The if-else statement is a two way branching. The general syntax of simple **if - else** statement is:

  if (*condition*)

    statement_to_execute_if_condition_is_true;
  else
    statement_to_execute_if_condition_is_false;

Where, *statement* may be a single statement, a block, or nothing, and the else statement is optional. The conditional statement produces a scalar result, i.e., an integer, character or floating point type.

It is important to remember that an if statement in C can execute only one statement on each branch (T or F). If we desire that multiple statements be executed on a branch, we must **block** them inside of a **{** and **}** pair to make them a single **compound statement**. Thus, the C code for the flowchart segment above would be:

Flowchart Segment:
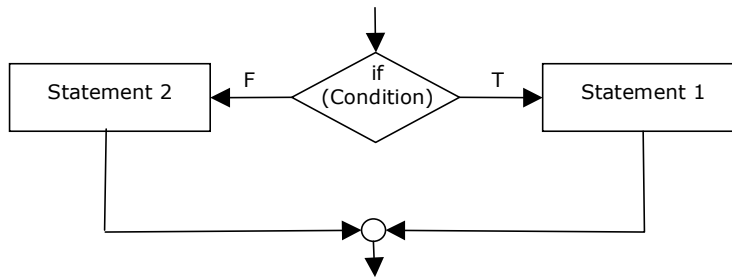
Example:

```
main()
{
        int num;
        printf(" Enter a number : ");
        scanf("%d",&num);
        if (num % 2  == 0)
                printf(" Even Number ");
        else
                printf(" Odd Number ");
}
```

2.1.3.   Nested if statement:

The ANSI standard specifies that 15 levels of nesting must be supported. In C, an else statement always refers to the nearest if statement in the same block and not already associated with if.

Example:

```
main()
{
        int num;
        printf(" Enter a number : ");
        scanf("%d",&num);
        if( num > 0 )
        {
                if( num % 2  == 0)
                        printf("Even Number");
                else
                        printf("Odd Number");
        }
        else
        {
                if( num < 0 )
                        printf("Negative Number");
                else
                        printf(" Number is Zero");
        }
}
```

Flowchart Segment:
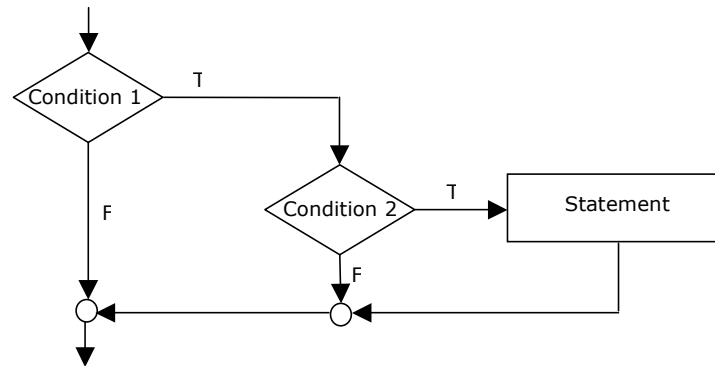
2.1.4. if-else-if Ladder:

When faced with a situation in which a program must select from *many* processing alternatives based on the value of a single variable, an analyst must expand his or her use of the basic selection structure beyond the standard two processing branches offered by the `if` statement to allow for multiple branches. One solution to this is to use an approach called **nesting** in which one (or both) branch(es) of a selection contain another selection. This approach is applied to each branch of an algorithm until enough additional branches have been created to handle each alternative. The general syntax of a nested if statement is:

> if (*expression*)
> > *statement₁*
>
> else if (*expression*)
> > *statement₂*
> > ..
> > ..
> else
> > *statement₃*

Example:

```
#include <stdio.h>
void main (void)
{
        int N;                          /* Menu Choice */
        printf ("MENU OF TERMS\n\n");
        printf ("1. Single\n");
        printf ("2. Double\n");
        printf ("3. Triple\n");
        printf ("4. Quadruple\n\n");
        printf ("Enter the numbe (1-4): ");
        scanf ("%d", &N);
        if (N == 1) printf ("one");
                else if (N == 2) printf ("two");
                        else if (N == 3) printf ("three");
                                else if (N == 4) printf ("four");
                                        else printf ("ERROR");

}
```
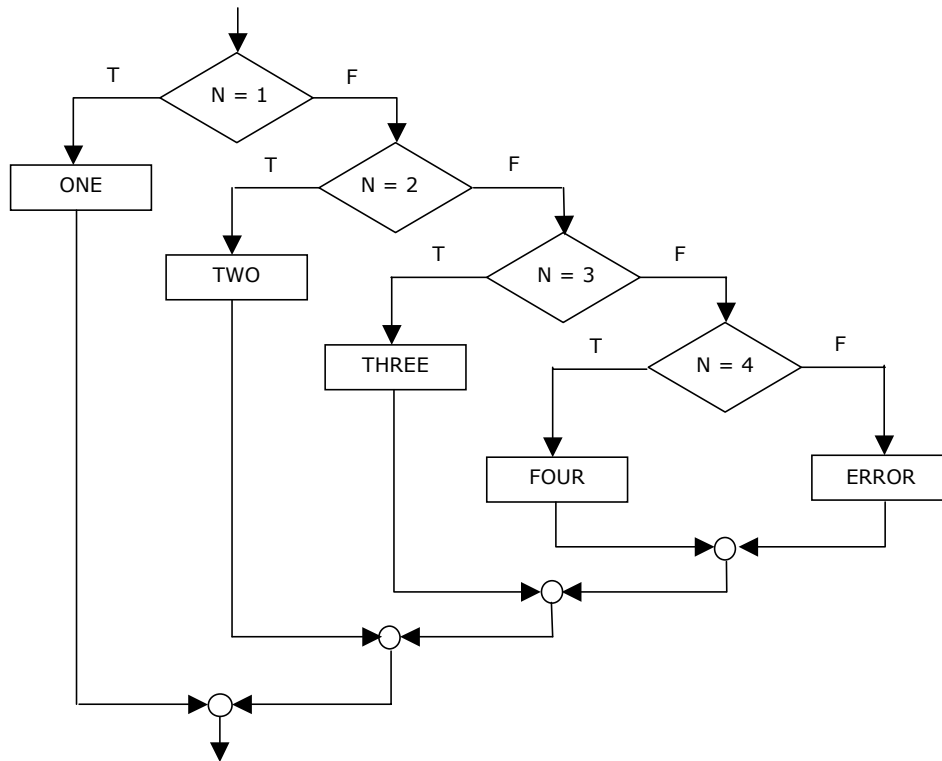
2.2. The ? : operator (ternary):

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

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

It simply states as:

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

Example:

Assign the maximum of a and b to z:

```
main()
{
        int a,b,z;
        printf("\n Enter  a and b ");
        scanf("%d%d",&a,&b);
        z = (a > b) ? a : b;
        printf("Maximum number: %d", z);
}
```

which is the same as:

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

2.3. The switch case statement:

The switch-case statement is used when an expression's value is to be checked against several values. If a match takes place, the appropriate action is taken. The general form of switch case statement is:

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

        case constant₂:
        statement;
        break;

        default:
        statement;
        break;
}
```

In this construct, the expression whose value is being compared may be any valid expression, including the value of a variable, an arithmetic expression, a logical comparison rarely, a bit wise expression, or the return value from a function call, but not a floating-point expression. The expression's value is checked against each of the specified cases and when a match occurs, the statements following that case are executed. When a break statement is encountered, control proceeds to the end of the switch - case statement.

The break statements inside the switch statement are optional. If the break statement is omitted, execution will continue on into the next case statements even though a match has already taken place until either a break or the end of the switch is reached.

The keyword case may only be constants, they cannot be expressions. They may be integers or characters, but not floating point numbers or character string.

Case constants may not be repeated within a switch statement.

The last case is a special keyword default. The default statement is executed if no matches are found. The default is optional and if it is not present, no action takes place if all matches fail.

Three important things to know about switch statement:

1. The switch differs from the if in that switch can only test for equality whereas if can evaluate any type of relational or logical expression.

2. No two case constants in the same switch can have identical values. But, a switch statement enclosed by an outer switch may have case constants and either same.

3. If character constants are used in the switch statement, they are automatically converted to integers.

Flowchart Segment - Case Selection:

In the example below, five possible paths might be followed depending on the value stored in the character storage location X. Each path is selected based on the individual value(s) that might be stored in X.

```
                    ┌──────────────┐
                    │   switch     │
                    │ (expression) │
                    └──────┬───────┘
                         N │
                    ╱──────▼──────╲       Y   ┌───────────┐      ┌───────────┐
                   ╱   case 1      ╲─────────▶│ Statement │─────▶│   Break   │──────▶
                   ╲   match ?     ╱          └───────────┘      └───────────┘
                    ╲──────┬──────╱
                         N │
                    ╱──────▼──────╲       Y   ┌───────────┐      ┌───────────┐
                   ╱   case 2      ╲─────────▶│ Statement │─────▶│   Break   │──────▶
                   ╲   match ?     ╱          └───────────┘      └───────────┘
                    ╲──────┬──────╱
                         N │
                    ╱──────▼──────╲       Y   ┌───────────┐      ┌───────────┐
                   ╱   case n      ╲─────────▶│ Statement │─────▶│   Break   │──────▶
                   ╲   match ?     ╱          └───────────┘      └───────────┘
                    ╲──────┬──────╱
                         N │
                    ┌──────▼───────┐
                    │   Default    │
                    │  Statement   │
                    └──────┬───────┘
                    ┌──────▼───────┐
                    │    Next      │
                    │  Statement   │
                    └──────────────┘
```
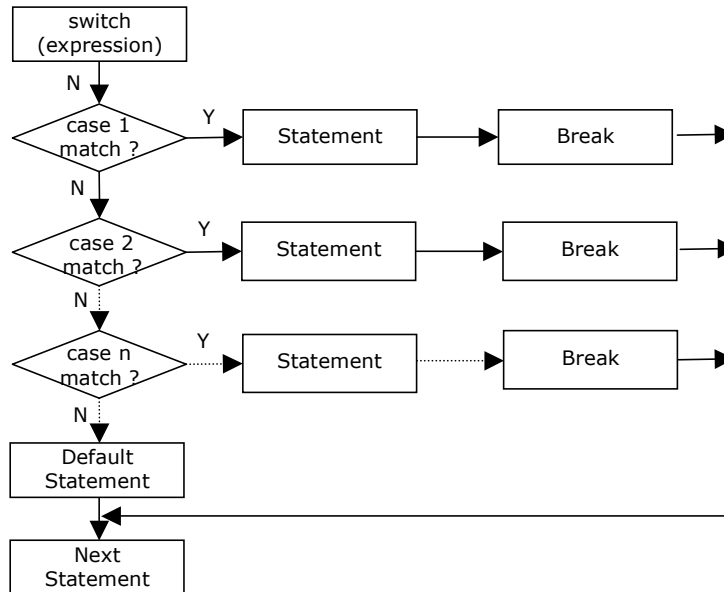
Example 1:

```
main()
{
        char gender;
        printf ("Enter Gender code:(M/F)");
        scanf ("%c", &gender);
        switch (gender)
        {
                case 'M' : printf (" Male");
                            break;
                case 'F'  : prrintf ("Female");
                            break;
                default :  printf ("Wrong code");
        }
}
```

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

Example 2:

```
        switch (letter)
        {
                case `A':
                case `E':
                case `I' :
```

```
                case `O':
                case `U':
                        numberofvowels++;
                        break;
                case ` ':
                        numberofspaces++;
                        break;
                default:
                        numberofconstants++;
                        break;
        }
```

In the above example if the value of letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented. If the value of letter is ` ' then numberofspaces is incremented. If none of these is true then the default condition is executed, that is numberofconstants is incremented.

2.4.   Looping and Iteration:

Looping is a powerful programming technique through which a group of statements is executed repeatedly, until certain specified condition is satisfied. Looping is also called a repetition or iterative control mechanism.

C provides three types of loop control structures. They are:

- for statement
- while statement
- do-while statement

2.4.1.   The for statement:

The for loop statement is useful to repeat a statement/s a known number of times. The general syntax is as follows:

```
    for (initialization; condition; operation)
            statement;
```

The **initialization** is generally an assignment statement that is used to set the loop control variable.

The **condition** is an expression(relational/logical/arithmetic/bitwise ….) that determines when the loop exists.

The **Operation** defines how the loop control variable changes each time the loop is repeated.
We must separate these three major sections by semicolon.

The for loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the for. The control flow of the for statement is as follows:

Example 1:

```
// printing all odd and even numbers between 1 to 5
        int x;
        main ()
        {
                for (x=1; x <=5 ; x++)
                {
                        if( x % 2 == 0 )
                                printf( " %d is EVEN \n",x);
                        else
                                printf(" %d is ODD \n",x);
                }
        }
```

Output to the screen:

1 is ODD
2 is EVEN
3 is ODD
4 is EVEN
5 is EVEN

Example 2:

```
//  sum the squares of all the numbers between 1 to 5
main()
{
        int x, sum = 0;
        for (x = 1; x  <= 5; x ++)
        {
                sum = sum +  x * x;
        }
        printf ("\n Sum of squares of all the numbers between 1 to 5 = %d ", sum);
}
```

Output to the screen:

Sum of squares of all the numbers between 1 to 5 = 55

Flowchart Segment - for Statement:
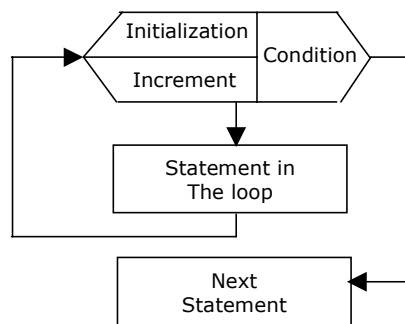
The **comma (,) operator** is used to extend the flexibility of the for loop. It allows the general form to be modified as follows:

for (initialization_1, initialization_2; condition; operation_1, operation_2)
statement;

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

1.   for (x=0; ((x>3) && (x<9)); x++)
2.   for (x=0,y=4; ((x>3) && (y<9)); x++, y+=2)
3.   for (x=0, y=4, z=4000; z; z/=10)

The second example shows that multiple expressions can be separated by a , (comma).

Example:

```
main()
{
        int j ;
        double  degC, degF;
        clrscr ();
        printf ("\n Table of Celsius  and Fahrenheit  degrees \n\n");
        printf ("Celsius Degree \t  Fahrenheit Degree \n")
        degC = -20.0;
        for (j = 1; j <= 6; j++)
        {
                degC = degC + 20.0;
                degF = (degC * 9.0/5.0) + 32.0;
                printf ("\n %7.2lf\t\ %7.2lf ", degC, degF);
        }
}
```

Output:

Table of Celsius and Fahrenheit degrees
Celsius Degree   Fahrenheit Degree
0.00   32.00
20.00   68.00
40.00   104.00
60.00   140.00
80.00   176.00
100.00   212.00

2.4.2.   Nested for loop:

Nested loops consist of one loop placed inside another loop. An example of a nested for loop is:

```
for (initialization; condition; operation)
{
        for (initialization; condition; operation)
        {
                statement;
        }
        statement;
}
```

In this example, the inner loop runs through its full range of iterations for each single iteration of the outer loop.

Example:

Program to show table of first four powers of numbers 1 to 9.

```c
#include <stdio.h >

void main()
{
        int i, j, k, temp;
        printf("I\tI^2\tI^3\tI^4 \n");
        printf("-------------------------------\n");
        for ( i = 1; i < 10; i ++)              /* Outer loop */
        {
                for (j = 1; j < 5; j ++)                /* 1st  level of nesting */
                {
                        temp = 1;
                        for(k = 0; k < j; k ++)
                                temp = temp * I;
                        printf ("%d\t", temp);
                }
                printf ("\n");
        }
}
```

Output to the screen:

I	I^2	I^3	I^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

2.4.3.   Infinite for loop:

We can make an endless loop by leaving the conditional expression empty as given below:
```c
        for( ; ; )
                printf("This loop will run for ever");
```

To terminate the infinite loop the break statement can be used anywhere inside the body of the loop. A sample example is given below:

```
for(; ;)
{
        ch = getchar ();
        if(ch == 'A')
                break;
}
printf("You typed an A");
```

This loop will run until the user types an A at the keyboard.

2.4.4.   for with no bodies:

A C-statement may be empty. This means that the body of the for loop may also be empty. There need not be an expression present for any of the sections. The expressions are optional.

Example 1:

```
/*   The loop will run until the user enters 123  */

        for( x = 0; x != 123; )
                scanf ("%d", &x);
```

This means that each time the loop repeats, 'x' is tested to see if it equals 123, but no further action takes place. If you type 123, at the keyboard, however the loop condition becomes false and the loop terminates.

The initialization some times happens when the initial condition of the loop control variable must be computed by some complex means.

Example 2:

```
/* Program to print the name in reverse order. */

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

void main()
{
        char s[20];
        int x;
        clrscr ();
        printf ("\nEnter your name: ");
        gets (s);
        x = strlen (s);
        for ( ; x > 0 ; )
        {
                --x;
                printf ("%c\t", s[x]);
        }
}
```

Output to the screen:

Enter your name: KIRAN

N   A   R   I   K

2.5. The while statement:

The second loop available in 'C' is while loop.

The general format of while loop is:

> while (*expression*)
>   *statement*

A while statement is useful to repeat a statement execution as long as a condition remains true or an error is detected. The while statement tests the condition before executing the statement.

The condition, can be any valid C languages expression including the value of a variable, a unary or binary expression, an arithmetic expression, or the return value from a function call.

The statement can be a simple or compound statement. A compound statement in a while statement appears as:

> while (condition)
> {
>   statement1;
>   statement2;
> }

With the if statement, it is important that no semicolon follow the closing parenthesis, otherwise the compiler will assume the loop body consists of a single null statement. This usually results in an infinite loop because the value of the condition will not change with in the body of the loop.

Example:

```
main()
{
        int j = 1;
        double  degC, degF;
        clrscr ();
        printf ("\n Table of Celsius  and Fahrenheit  degrees \n\n");
        printf ("Celsius Degree \t  Fahrenheit Degree \n")
        degC = -20.0;
        while (j <= 6)
        {
                degC = degC + 20.0;
                degF = (degC * 9.0/5.0) + 32.0;
                printf ("\n %7.2lf\t\ %7.2lf ", degC, degF);
                j++;
        }
}
```

Output:

Table of Celsius and Fahrenheit degrees
Celsius Degree   Fahrenheit Degree
0.00   32.00
20.00   68.00
40.00   104.00
60.00   140.00
80.00   176.00
100.00   212.00

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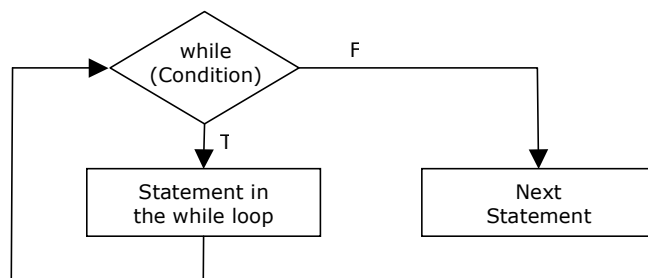
Because the while loop can accept expressions, not just conditions, the following are all legal:

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

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

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

        while(i++ < 10);

The counts i up to 10.

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

This uses C standard library functions: getchar () to reads a character from the keyboard and putchar () to writes a given char to screen. The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read.

Nested while:

Example:

Program to show table of first four powers of numbers 1 to 9.

#include <stdio.h >

```c
void main()
{
        int i, j, k, temp;
        printf("I\tI^2\tI^3\tI^4 \n");
        printf("-----------------------------\n");
        i = 1;
        while (i < 10)                          /* Outer loop */
        {
                j = 1;
                while (j < 5)                   /* 1st  level of nesting */
                {
                        temp = 1;
                        k = 1;
                        while (k < j)
                        {
                                temp = temp * i;
                                k++;
                        }
                        printf ("%d\t", temp);
                        j++;
                }
                printf ("\n");
                i++;
        }
}
```

Output to the screen:

I	I^2	I^3	I^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

2.5.1.   The do-while statement:

The third loop available in C is do – while loop.

The general format of do-while is:

        do
                statement;
        while (*expression*);

Unlike for and while loops, which tests the condition at the top of the loop. The do – while loop checks its condition at the bottom of the loop. This means that the do – while loop always executes first and then the condition is tested. Unlike the while construction, the do – while requires a semicolon to follow the statement's conditional part.

If more than one statement is to be executed in the body of the loop, then these statements may be formed into a compound statement as follows:

```
do
{
        statement1;
        statement2;
} while (condition);
```

Flowchart Segment of do-while Statement:
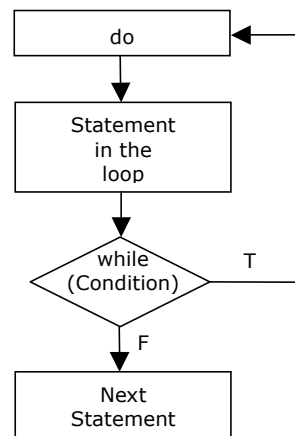
Example 1:

```
# include <stdio.h>
main()
{
        do
        {
                printf("x = %d\n", x--);
        } while(x > 0);
}
```

Output to the screen:

X = 3
X = 2
X = 1

Example 2:

```
#include <stdio.h>
void main()
{
        char ch;
```

```c
        printf("T: Train\n");
        printf("C: Car\n");
        printf("S: Ship\n");
        do
        {
                printf("\nEnter your choice: ");
                fflush(stdin);
                ch = getchar();
                switch(ch)
                {
                        case 'T' :
                                        printf("\nTrain");
                                        break;
                        case 'C' :
                                        printf("\nCar");
                                        break;
                        case 'S':
                                        printf("\nShip");
                                        break;
                        default:
                                        printf("\n Invalid Choice");
                }
        } while(ch == 'T' || ch == 'C' || ch == 'S');
}
```

Output to the screen:

> T: Train
> C: Car
> S: Ship

Enter your choice: T
      Train

Distinguishing between while and do-while loops:

While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

2.6.   Un-conditional (Jump) statements:

C has four jump statements to perform an unconditional branch:

- return
- goto
- break and
- continue

2.6.1 return statement:

A return statement is used to return from a function. A function can use this statement as a mechanism to return a value to its calling function. If now value is specified, assume that a garbage value is returned (some compilers will return 0).

The general form of return statement is:

return expression;

Where expression is any valid rvalue expression.

Example:

```
return x; or return(x);
return x + y or return(x + y);
return rand(x); or return(rand(x));
return 10 * rand(x); or return (10 * rand(x));
```

We can use as many return statements as we like within a function. However, the function will stop executing as soon as it encounters the first return. The } that ends a function also causes the function to return. It is same way as return without any specified value.

A function declared as void may not contain a return statement that specifies a value.

2.6.2. goto statement:

goto statement provides a method of unconditional transfer control to a labeled point in the program. The goto statement requires a destination label declared as:

label:

The label is a word (permissible length is machine dependent) followed by a colon. The goto statement is formally defined as:

goto label;
`
`
`
label:
target statement

Since, C has a rich set of control statements and allows additional control using break and continue, there is a little need for goto. The chief concern about the goto is its tendency to render programs unreachable. Rather, it a convenience, it used wisely, can be a benefit in a narrow set of programming situation. So the usage of goto is highly discouraged.

Example:

```
Void main()
{
    int x = 6, y = 12;

    if( x == y)
```

```
                        x++;
                else
                        goto error;
                error:
                        printf ("Fatal error; Exiting");
        }
```

The compiler doesn't require any formal declaration of the label identifiers.

2.6.3.   break statement:

We can use it to terminate a case in a switch statement and to terminate a loop.

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

Example:

```
void main()
{
        int value;

        while (scanf("%d", &value ) == 1 && value != 0)
        {
                if(value < 0)
                {
                        printf ("Illegal value\n");
                        break;                          /* Terminate the loop */
                }
                if(value > 100)
                {
                        printf("Invalid value\n");
                        continue;                       /* Skip to start loop again */
                }
        }                                               /* end while value != 0 */
}
```


2.6.4.   Continue statement:

The continue statement forces the next iteration of the loop to take place, skipping any code in between. But the break statement forces for termination.

Example 1:

/* Program to print the even numbers below 100 */

#include<stdio.h>

```
void main()
{
        int x;
```

```
        for(x = 1; x < 10; x++)
        {
                if (x % 2)
                        continue;
                printf ("%d\t", x)
        }
}
```

An odd number causes continue to execute and the next iteration to occur, by passing the printf () statement. A continue statement is used within a loop (i.e for, while, do – while) to end an iteration in while and do-while loops, a continue statement will cause control to go directly to the conditional test and then continue the looping process. In the case of for, first the increment part of the loop is performed, next the conditional test is executed and finally the loop continues.

Example 2:

```
main()
{
        char ch;
        while (1)
        {
                ch = getchar();
                if (ch == EOF)
                        break;
                if (iscntrl (ch))
                        continue;
                else
                        printf ("\n not a control character");
}
```

Distinguishing between break and continue statement:

Break	Continue
Used to terminate the loops or to exist loop from a switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop containing it.	The continue statement when executed cause immediate termination of the current iteration of the loop.

2.6.5.   The exit () function:

Just as we can break out of a loop, we can break out of a program by using the standard library function exit(). This function causes immediate termination of the entire program, forcing a return to the operation system.

The general form of the exit() function is:

void exit (int return_code);

The value of the return_code is returned to the calling process, which is usually the operation system. Zero is generally used as a return code to indicate normal program termination.

Example:

```
Void menu(void)
{
        char ch;
        printf("B: Breakfast");
        printf("L: Lunch");
        printf("D: Dinner");
        printf("E: Exit");
        printf("Enter your choice: ");
        do
        {
                ch = getchar();
                switch (ch)
                {
                        case 'B' :
                                printf ("time for breakfast");
                                break;
                        case 'L' :
                                printf ("time for lunch");
                                break;
                        case 'D' :
                                printf ("time for dinner");
                                break;
                        case 'E' :
                                exit (0);          /* return to operating system */
                }
        } while (ch != 'B' && ch != 'L' && ch != 'D');
}
```

2.7.   ARRAYS:

An array is a collection of variables of the same type that are referenced by a common name. In C, all arrays consists of contiguous memory locations. The lowest address corresponds to the first element, and the highest address to the last element. Arrays may have from one to several dimensions. A specific element in an array is accessed by an index.

One Dimensional Array:

The general form of single-dimension array declaration is:

        Type variable-name[size];

Here, type declares the base type of the array, size defines how many elements the array will hold.

For example, the following declares as integer array named sample that is ten elements long:

        int sample[10];

In C, all arrays have zero as the index of their first element. This declares an integer array that has ten elements, sample[0] through sample[9]

Example 1:

/* load an inter array with the number 0 through 9 */

```
main()
{
        int x [10], t;
        for (t=; t<10; t++)
                x [t] = t;
}
```

Example 2:

/* To find the average of 10 numbers */
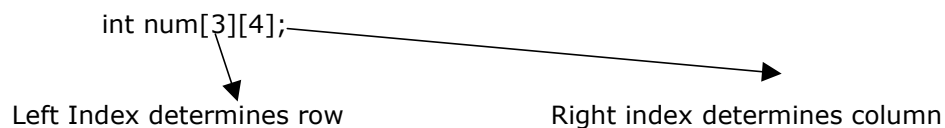
```
# include <stdio.h>
main()
{
        int i, avg, sample[10];

        for (i=0; i<10; i++)
        {
                printf ("\nEnter number: %d ", i);
                scanf ("%d", &sample[i]);
        }
        avg = 0;
        for (i=0; i<10; i++)
                avg = avg + sample[i];
        printf ("\nThe average is: %d\n", avg/10);
}
```

Two-dimensional arrays:

To declare two-dimensional integer array num of size (3,4),we write:

int num[3][4];

Left Index determines row   Right index determines column

Two dimensional arrays are stored in a row-column matrix where the first index indicates the row and the second indicates the column. This means that the right most index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

Num[t][I]	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Example:

```
main ()
{
        int t, i, num [3][4];

        for (t=0; t<3; t++)
                for (i=0; i<4; ++i)
                        num [t][i] = (t * 4) + i + 1;
        for (t=0; t<3; t++)
        {
                for (i=0; i<4; ++i)
                        printf ("%3d", num[t][i]);
                printf ("\n");
        }
}
```

the graphic representation of a two-dimensional array in memory is:

byte = sizeof 1st Index * sizeof 2nd Index * sizeof (base type)

Size of the base type can be obtained by using **size of operation**.

returns the size of memory (in terms of bytes) required to store an integer object.

sizeof (unsigned short)	=	2
sizeof (int)	=	4
sizeof (double)	=	8
sizeof (float)	=	4

assuming 2 byte integers as integer with dimension 4, 3 would have

$4 * 3 * 2 = 24$ bytes

Given: char ch[4][3]

Ch [0][0]	Ch [0][1]	Ch [0][2]
Ch [1][0]	Ch [1][1]	Ch [1][2]
Ch [2][0]	Ch [2][1]	Ch [2][2]
Ch [3][0]	Ch [3][1]	Ch [3][2]

N - dimensional array or multi dimensional array:

This type of array has n size of rows, columns and spaces and so on. The syntax used for declaration of this type of array is as follows:

Data type   array name[s1] [s2] … … … [sn];

In this sn is the nth size of the array.

Array Initialization:

The general form of array initialization is:

Type_specifier array_name[size1]…. [sizeN] = { value_list};
The value list is a comma_separated list of constants whose type is compatible with type_specifier.

Example 1:

10 element integer array is initialized with the numbers 1 through 10 as:

      int I[10] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

      i.e., I[0] will have the value 1 and

                  ……….

                  ……….
      I[9] will have the value 10

Character arrays that hold strings allow a shorthand initialization that takes the form:

      char array_name[size] = "string" ;

Example 2:

      char str[9] = "I like C";

this is same as writing:

      char str[9] = {'I', ' ', 'l', 'i', 'k', 'e', ' ', 'C', '\0'};

2.8. STRINGS:

In C, In order to allow variable length strings the \0 character is used to indicate the end of a string. A null is specified using '\0' and is zero. Because of the null terminator, It is necessary to declare character array to be one character longer than the largest string that they are to hold. For example, to declare an array str that can hold a 10 character string, we write:

      char str[11];

this makes room for the null at the end of the string.

A string constant is a list of characters enclosed by double quotation marks.
For example:

      "hello, this is a test"

It is not necessary to manually add the null onto end of string constants – the compiler does this automatically.

T	U	R	B	O	C	\0'

Reading a string from the keyboard:

The easiest way to input a string from the keyboard is with the gets() library function. The general form gets() is:

        gets(array_name);

To read a string, call gets() with the name of the array, with out any index, as its arguments. Upon return form gets() the array will hold the string input. The gets() function will continue to read characters until you enter a carriage return. The header file used for gets() is stdio.h

Example:

```
# include <stdio.h>
main()
{
        char str[80];
        printf ("\nEnter a string:");
        gets (str);
        printf ("%s", str);
}
```

the carriage return does not become part of the string instead a null terminator is placed at the end.

Writing strings:

The puts() functions writes its string argument to the screen followed by a newline. Its prototype is:

        puts(string);

It recognizes the same back slash code as printf(), such as "\t" for tab. As puts() can output a string of characters – It cannot output numbers or do format conversions it required faster overhead than printf(). Hence, puts() function is often used when it is important to have highly optimized code.

For example, to display "hello" on the screen:

        puts("hello");

Array of strings:

To create an array of strings, a two dimensional character array is used with the size of the left-Index determining the number of strings and the size of the right Index specifying the maximum length of each string.

For example, to declare an array of 30 strings each having a max length of 80 characters.

        char str_array[30][80];

To access an individual string is quite easy: you simply specify only the left Index.

Example:

/* to accept lines of text and redisplay them when a blank line is entered */

```
main()
{
       int t, i;
       char text [100][80];
       for (t=0; t<100; t++)
       {
              printf ("%d Enter Text: ", t);
              gets (text [t]);
              if (! text [t][0])              /* quit on blank line */
                     break;
       }
       for (i=0; i<t; i++)
              printf ("%s\n", text [i]);
}
```

2.8.1   Basic String Handling Functions:

As string data type is not present. A string constant is a list of characters enclosed in double quotes.

For example, "Hello"

C- supports a wide range of string manipulation functions,

Name	Function
From to strcpy(s1, s2)	Copies s2 into s1 (the array forming to must be large enough to hold the string content in form)
strcat(s1, s2)	Append s2 onto the end of s1
strlen(s1)	Returns the length of s1
strcmp(s1, s2)	Returns 0 if s1 and s2 are the same to determine alphabetic order. Less than 0 if s1 < s2; greater than 0 if s1 > s2
strchr(s1, ch)	Return a pointer to the first occurrence of ch in s1
strstr(s1, s2)	Return a pointer to the first occurrence of s2 in s1
strrev(s1)	Reverses the string s1.

All the string handling functions are prototyped in: # include <string.h>

strcat () and strcopy () both return a copy of their first argument which is the destination array. Note the order of the arguments is **destination array** followed by **source array** which is sometimes easy to get the wrong around when programming.

The strcmp () function **lexically** compares the two input strings and returns:

Less than zero: if string1 is lexically less than string2

Zero: if string1 and string2 are lexically equal

Greater than zero: if string1 is lexically greater than string2

This can also confuse beginners and experience programmers forget this too.

The strcat (), strcmp () and strcpy () copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first n characters. Note the NULL terminated requirement may get violated when using these functions.

Example 1:

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

void main(void)
{
        char s1 [80], s2 [80];
        gets (s1);
        gets (s2);
        printf ("lengths: %d %d\n", strlen (s1), strlen (s2));
        if (! Strcmp (s1, s2))          /* strcmp () returns false if the strings are equal, */
                                        */ use ! to reverse the condition*/
                printf("two strings are equal \n");
        strcat (s1, s2);
        printf ("%s\n", s1);
        strcpy (s1,"this is a test\n");
        printf (s1);
        if (strchr ("hello", 'e')
                printf("e is in hello\n");
        if (strstr ("hi these", "hi");
                printf ("found hi");
}
Output to the screen:

enter: hello hello
lengths: 5 5
two strings are equal
hellohello
this is a test
e is in hello
found hi
```

Example 2:

```
/* to reverse a string * /

# include <stdio.h>
# include <string.h>
main ()
{
        char str[80];
        int i;
        printf ("enter a string: ");
        gets (str);
        for (i = strlen (str)-1; i > 0; i--)
                printf ("%c", str[i]);
}
```

2.8.2.   Character conversions and testing: ctype.h

We conclude this chapter with a related library #include <ctype.h> which contains many useful functions to convert and test **single** characters. The common functions are prototypes as follows:

Character testing:

int isalnum (int c) -- True if c is alphanumeric.

int isalpha (int c) -- True if c is a letter.

int isascii (int c) -- True if c is ASCII .

int iscntrl (int c) -- True if c is a control character.

int isdigit (int c) -- True if c is a decimal digit

int isgraph (int c) -- True if c is a graphical character.

int islower (int c) -- True if c is a lowercase letter

int isprint (int c) -- True if c is a printable character

int ispunct (int c) -- True if c is a punctuation character.

int isspace (int c) -- True if c is a space character.

int isupper (int c) -- True if c is an uppercase letter.

int isxdigit (int c) -- True if c is a hexadecimal digit

2.8.3.   Character Conversion:

int toascii (int c) -- Convert c to ASCII .

tolower (int c) -- Convert c to lowercase.

int toupper (int c) -- Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

Glossary and Keywords

The ANSI standard categorizes C's program control statements as follows:

Category	Relevant Keywords
Selection (or Conditional) Statements	if switch
Iteration	for while do-while
Jump	break goto return continue
Label	case default

Conditional Expressions in C:

Many C statements rely on the outcome of a conditional expression, which evaluates to either **true** or **false**. In C, unlike many other languages, true is any non-zero value and false is zero. This approach facilitates efficient coding.

The programmer is not restricted to conditional expressions involving only the relational and logical operators. Any valid C expression may be used to control the if. All that is required is an evaluation to either zero or no-zero. For example:

```
b = a * c;

if (b)
        printf (&quot%d\n&quot,b);
else
        printf ("b evaluated to zero\n&quot);
```

The if Statement:

The general form of an if statement is:

```
if (expression)
        statement;
else
        statement;
```

where, *statement* may be a single statement, a block, or nothing, and the else statement is optional. The conditional statement produces a scalar result, ie, an integer, character or floating point type.

Nested ifs. The ANSI standard specifies that 15 levels of nesting must be supported. In C, an else statement always refers to the nearest if statement in the same block and not already associated with an if. For example:

```
if(i)
{
        if(j)
                statement 1;
        if(k)                   // this if is associated with
                statement 2;
        else                    // this else
```

```
                        statement 3;
        }
        else                        // this is associated with if(i)
                statement 4;
```
The if-else-if Ladder. To avoid excessive indentation, if-else-if ladders are normally written like this:

```
        if(expression)
                statement;
        else if(expression)
                statement;
        else if(expression)
                statement;
                ...
        else
                statement;
```

The switch Statement:

switch successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The general form is:

```
        switch (expression)
        {
                case constant1:
                        statement sequence;
                        break;
                case constant2:
                        statement sequence;
                        break;
                        ...

                default:
                        statement sequence;
        }
```

default: is executed if no matches are found.

Although case is a label statement, it cannot exist outside a switch.
Some important aspects of switch are as follows:

- A switch can only test for equality, whereas if can evaluate an expression.
- No two case constants can have identical values.
- If character constants are used, they will automatically be converted to integers.

If the break statement is omitted, execution continues on into the next case statement until either a break or the end of the switch is reached.

Nested switch Statements. A switch statement may be part of the statement sequence of an outer switch.

Iteration Statements:

The for Loop:

The general form of the for statement is:

```
For (initialization; condition; operation)
        statement;
```

initialization is generally an assignment statement used to set the loop control variable. *condition* is a relational expression that determines when the loop exits. operation defines how the loop variable changes each time the loop is repeated.

In for loops, the conditional test is always performed at the top of the loop, which means that the code inside the loop may not be executed at all if the condition is false, to begin with as in:

```
x = 10;
for (y=10; y != x; ++y)
        printf (" %d", y);
```

Variation 1 - The Comma Operator. A variant of the for loop is made possible by the comma operator, as in:

```
for(x=0, y=0; x+y < 10; ++x);
```

in which both x and y control the loop.

Variation 2 - Missing Pieces of the Loop Definition. An interesting trait of the for loop is that pieces of the loop definition need not be there. For example, in:

```
for (x=0; x!=123; )
        scanf ("%d", &x);
```

each time the loop repeats, x is tested to see if it equals 123. The loop condition only becomes false, terminating the loop, when 123 is entered.

Variation 3 - The Infinite Loop. If all of the pieces in the loop definition are missing, an infinite loop is created. However, the break statement may be used to break out of the loop, as in:

```
for(;;)
{
        ch = getchar();
        if(ch == 'A')
                break;
}
```

Variation 4 - for Loops with No Bodies. The body of the for loop may also be empty. This improves the efficiency of some code. For example, the following removes leading spaces from the stream pointed to by str:

```
for ( ; *str==' '; str++) ;
```

Time delay loops are another application of a loop with an empty body, eg:
```
for (t=0; t<1000; t++);
```

The while Loop:

The general form of the while loop is:

        while (*condition*)
                statement;

where *statement* is either an empty statement, a single statement or a block of statements. The loop iterates while the condition, which is executed at the top of the loop, is true.
Variation 1 - No Body. The following, which is an example of a while loop with no body, loops until the user types A:

        while ((ch = getchar ()) !='A') ;

The do-while Loop:

The do-while loop tests the condition at the bottom of the loop rather than at the top. This ensures that the loop will always execute at least once. In the following example, the loop will read numbers from the keyboard until it finds a number equal to or less than 100:

        do
        {
                scanf("%d", &num);
        } while (num>100);

Jump Statements:

The return Statement:

The return statement is used to return from a function. If return has a value associated with it, that value is returned by the function. If no return value is specified, either zero or a garbage value will be returned, depending on the compiler. The general form of the return statement is:

        return *expression*;

The **}** which ends a function also causes the function to return. A function declared as void may not contain a return statement that specifies a value

The goto Statement:

The goto statement requires a label (an identifier followed by a colon), which must be in the same function as the goto.

The break Statement:

The break statement has two uses:

- It can be used to terminate a case in a switch.
- It can be used to force immediate termination of a loop

The exit () Function:

The standard library function exit () causes immediate termination of the entire program. The general form of the exit function is:

void exit (int *return_code*);

The value of *return_code* is returned to the operating system. Zero is generally used as a return code to indicate normal program termination. Other arguments are used to indicate some sort of error.

The continue Statement:

The continue statement forces the next iteration of the loop, skipping any code in between. For the for loop, continue causes the conditional test and then the increment portion of the loop to execute. For the while and do-while loops, program control passes to the conditional test.

ARRAYS AND STRINGS:

An **array** is a collection of variables of the same type which are referenced by a common name. A specific element in an array is referenced by an **index**. The most common array in C is the string, which is simply an array of characters terminated by a null.

In C, arrays and pointers are closely related; a discussion of one usually refers to the other.

C has no bounds checking on arrays.

Single Dimension Arrays:

The general form for declaring a **single-dimension array** is:

type var_name[size];

In C, all arrays have zero as the index of their first element. Therefore a declaration of char p[10]; declares a ten-element array (p[0] through p[9]).

Generating a Pointer to an Array:

A pointer to the first element in an array may be generated by simply specifying the array name, without any index. For example, given:

int sample[10];

a pointer to the first element may be generated by simply using the name sample. For example, the following code fragment assigns p the address of the first element of sample:

```
int *p;
int sample[10];

p = sample;
```

The address of the first element may also be specified using the & operator. For example, sample and &sample[0] produce the same result. The former is usually used.

Passing Single-Dimension Arrays to Functions:

In C, an entire array cannot be passed as an argument to a function. However, a *pointer* to an array may be passed by specifying the array's name without an index. If a function receives a single dimension array, the formal parameter may be declared as either a pointer, a sized array, or an unsized array. Examples are:

function(int x)   /* A pointer */

function(int x[10])   /* A sized array */

function(int x[])   /* An unsized array */

Strings:

A string is actually an array of characters. Because strings are terminated by a null ('\0'), character arrays must be declared with one extra element (to hold the null).

Although C does not have a string data type, it allows string constants. For example, "hello there" is a string constant.

C supports a wide range of string manipulation functions, including:

Function	Description
strcpy (s1, s2)	Copies s2 into s1.
strcat (s1, s2)	Concatenates s2 to s1.
strlen (s1)	Returns the length of s1.
strcmp (s1, s2)	Returns 0 (false) if s1 and s2 are the same. Returns less than 0 if s1&lts2 Returns greater than 0 if s1>s2
strchr (s1, ch)	Returns pointer to first occurrence ch in s1.
strstr (s1, s2)	Returns pointer to first occurrence s2 in s1.
strrev (s1)	Reverses the string s1.

Since strcmp () returns false if the strings are equal, it is best to use the ! operator to reverse the condition if the test is for equality.

Two-Dimensional Arrays:

In C, a **two-dimensional array** is declared as shown in the following example:

int d[10][20];

Two-dimensional arrays are stored in a row-column matrix. The first index indicates the row. The second index indicates the column.

When a two-dimensional array is used as an argument to a function, only a pointer to the first element is passed. However, the receiving function must define at least the length of the second dimension.

Example:

```
function (int x[][20]);
```


Arrays of Strings:

Arrays of strings are created using a two-dimensional array. The left index determines the number of strings. Each string is accessed using only the left index.

Multi-Dimensional Arrays:

C allows **arrays of more than two dimensions**, the exact limit depending on the individual compiler.

Indexing Pointers:

In C, pointers and arrays are closely related. As previously stated, an array name without an index is a pointer to the first element. For example, given the array char

my_array[10], my_array and &my_array[0] are identical.

Conversely, any pointer variable may be indexed as if it were declared as an array. For example, in this program fragment:

```
int *p, i[10];

p = i;
p[5] = 100;              /* assignment using index */
(p+5) = 100              /* assignment using pointer arithmetic  */
```

both assignments achieve the same thing.

Pointers are sometimes used to access arrays because pointer arithmetic is faster than array indexing.

In a sense, a two-dimensional array is like an array of pointers that point to arrays of rows. Therefore, using a separate pointer variable is one easy way to access elements.

For example, the following prints the contents of the specified row for the global array num:

```
int num[10][10];
        ...
void print_row(int j)
{
        int *p, t;
        p = &num[j][0]; // get address of first element in row j
        for(t=0;t<10;++t)
        printf("%d ", *(p+t));
}
```


Array Initialization:

Arrays may be initialized at the time of declaration. The following example initializes a ten-element integer array:

```
int i[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

Character arrays which hold strings allow a shorthand initialization, e.g.:

```
char str[9] = "I like C";
```

which is the same as:

```
char str[9] = { 'I',' ','l', 'i', 'k', 'e',' ','C','\0' };
```

When the string constant method is used, the compiler automatically supplies the null terminator.

Multi-dimensional arrays are initialized in the same way as single-dimension arrays, e.g.:

```
        int sgrs[6][2] =
        {
                1, 1,
                2, 4,
                3, 9,
                4, 16,
                5, 25
                6, 36
        };
```

Unsized Array Initializations:

If unsized arrays are declared, the C compiler automatically creates an array big enough to hold all the initializes. This is called an unsized array.
Example:

declaration/initializations are as follows:

```
        char e1[] = "read error\n";
        char e2[] = "write error\n";

        int sgrs[][2] =
        {
                1, 1,
                2, 4,
                3, 9
                4, 16,
        };
```

SAMPLE C – PROGRAMS

1. Program to find whether a given year is leap year or not.

```c
# include <stdio.h>

main()
{
    int year;

    printf("Enter a year:\n");
    scanf("%d", &year);

    if ( (year % 4) == 0)
        printf("%d is a leap year", year);
    else
        printf("%d is not a leap year\n", year);
}
```

Output:

Enter a year: 2000
2000 is a leap year

RUN2:

Enter a year: 1999
1999 is not a leap year

2. Program to multiply given number by 4 using bitwise operators.

```c
# include <stdio.h>

main()
{
    long number, tempnum;
    printf("Enter an integer:\n");
    scanf("%ld", &number);
    tempnum = number;
    number = number << 2;   /*left shift by two bits*/

    printf("%ld x 4 = %ld\n", tempnum, number);
}
```

Output:

Enter an integer: 15
15 x 4 = 60

RUN2:
Enter an integer: 262
262 x 4 = 1048

3.   Program to compute the value of X ^ N given X and N as inputs.

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

void main()
{
      long int x, n, xpown;
      long int power(int x, int n);

      printf("Enter the values of X and N: \n");
      scanf("%ld %ld", &x, &n);

      xpown = power (x, n);

      printf("X to the power N = %ld\n");
}

/*Recursive function to computer the X to power N*/

long int power(int x, int n)
{
      if (n==1)
              return(x);
       else if ( n%2 == 0)
              return (pow(power(x,n/2),2));            /*if n is even*/
      else
              return (x*power(x, n-1));               /* if n is odd*/
}
```

Output:

Enter the values of X and N: 2 5
X to the power N = 32

RUN2:
Enter the values of X and N: 4 4
X to the power N ==256

RUN3:
Enter the values of X and N: 5 2
X to the power N = 25

RUN4:
Enter the values of X and N: 10 5
X to the power N = 100000

4.   Program to swap the contents of two numbers using bitwise XOR operation.
Don't use either the temporary variable or arithmetic operators.

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

```
main()
{
        long i, k;
        printf("Enter two integers: \n");
        scanf("%ld %ld", &i, &k);
        printf("\nBefore swapping i= %ld and k = %ld", i, k);
        i = i^k;
        k = i^k;
        i = i^k;
        printf("\nAfter swapping i= %ld and k = %ld", i, k);
}
```

Output:

Enter two integers: 23 34
Before swapping i= 23 and k = 34
After swapping i= 34 and k = 23

5.   Program to find and output all the roots of a quadratic equation, for non-zero coefficients. In case of errors your program should report suitable error message.

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

main()
{
        float A, B, C, root1, root2;
        float realp, imagp, disc;
        clrscr();
        printf("Enter the values of A, B and C\n");
        scanf("%f %f %f", &A,&B,&C);

        if( A==0 || B==0 || C==0)
        {
                printf("Error: Roots cannot be determined\n");
                exit(1);
        }
        else
        {
                disc = B*B - 4.0*A*C;
                if(disc < 0)
                {
                        printf("Imaginary Roots\n");
                        realp = -B/(2.0*A) ;
                        imagp = sqrt(abs(disc))/(2.0*A);
                        printf("Root1 = %f  +i %f\n", realp, imagp);
                        printf("Root2 = %f  -i %f\n", realp, imagp);
                }
                else if(disc == 0)
                {
                        printf("Roots are real and equal\n");
                        root1 = -B/(2.0*A);
                        root2 = root1;
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
```

```
                }
                else if(disc > 0 )
                {
                        printf("Roots are real and distinct\n");
                        root1 =(-B+sqrt(disc))/(2.0*A);
                        root2 =(-B-sqrt(disc))/(2.0*A);
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
        }

}
```

Output:

RUN 1
Enter the values of A, B and C: 3 2 1
Imaginary Roots
Root1 = -0.333333 +i 0.471405
Root2 = -0.333333 -i 0.471405

RUN 2
Enter the values of A, B and C: 1 2 1
Roots are real and equal
Root1 = -1.000000
Root2 = -1.000000

RUN 3
Enter the values of A, B and C: 3 5 2
Roots are real and distinct
Root1 = -0.666667
Root2 = -1.000000

6.   Write a C programme to accept a list of data items & find the II largest & II smallest in it & take average of both & search for that value. Display appropriate message on successful search.

```
main ()
{
        int i,j,a,n,counter,ave,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
        scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                for (j=i+1; j<n; ++j)
                {
                        if (number[i] < number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
```

```
        }
        printf ("The numbers arranged in descending order are given below\n");
        for (i=0; i<n; ++i)
                printf ("%10d\n",number[i]);
        printf ("The 2nd largest number is  = %d\n", number[1]);
        printf ("The 2nd smallest number is = %d\n", number[n-2]);
        ave = (number[1] +number[n-2])/2;
        counter = 0;
        for (i=0; i<n; ++i)
        {
                if (ave==number[i])
                ++counter;
        }
        if (counter==0)
                printf("The average of 2nd largest & 2nd smallest is not in the array\n");
        else
                printf("The average of 2nd largest & 2nd smallest in array is %d in
                        numbers\n", counter);
  }
```

7.   Write a C programme to arrange the given numbers in ascending order.

```
main ()
{
        int i,j,a,n,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] > number[j])
                                {
                                        a= number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
        }
        printf("Number in Asscending order:\n");
        for(i=0;i<n;i++)
                printf("\t%d\n",number[i]);
}
```

8.   Program to convert the given binary number into decimal.

```
# include <stdio.h>
main()
{
        int   num, bnum, dec = 0, base = 1, rem ;
        printf("Enter the binary number(1s and 0s)\n");
        scanf("%d", &num);          /*maximum five digits */
```

```c
        bnum = num;
        while( num > 0)
        {
                        rem = num % 10;
                        if(rem>1)
                        {
                                printf("\nError in input");
                                break;
                        }
                        dec = dec + rem * base;
                        num = num / 10 ;
                        base = base * 2;
        }
        if(num==0)
        {
                printf("The Binary number is = %d\n", bnum);
                printf("Its decimal equivalent is =%d\n", dec);
        }
}
```

9.   Program to generate the fibonacci sequence.

```c
#include <stdio.h>
main()
{
        int   fib1=0, fib2=1, fib3, limit, count=0;

        printf("Enter the limit to generate the fibonacci sequence\n");
        scanf("%d", &limit);

        printf("Fibonacci sequence is ...\n");
        printf("%d\n",fib1);
        printf("%d\n",fib2);
        count = 2;                      /* fib1 and fib2 are already used */

        while( count < limit)
        {
                fib3 = fib1 + fib2;
                count ++;
                printf("%d\n",fib3);
                fib1 = fib2;
                fib2 = fib3;
        }
}
```

10.   Program to reverse the given integer (palindrome).

```c
#include <stdio.h>
main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
```

```
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);

        if(temp == rev )
                printf("Number is a palindrome\n");
        else
                printf("Number is not a palindrome\n");
}
```

11.   Program to determine the given number is odd.

```
#include <stdio.h>

main()
{
        int numb;
        printf(" Enter the number\n");
        scanf("%d", &numb);

        if((numb%2)!=0)
                printf(" %d , is an odd number\n", numb);

}
```

12.   Program to find the largest among three numbers.

```
#include <stdio.h>

main()
{
        int a, b, c;
        printf(" Enter  the values for A,B,C\n");
        scanf("%d %d %d", &a, &b, &c);
        if( a > b )
        {
                if ( a > c)
                        printf(" A is the Largest\n");
                else
                        printf("C is the largest\n");
        }
        else if ( b > c)
                printf(" B is the Largest\n");
        else
                printf("C is the Largest\n");
}
```

13.   Program to find the areas of different geometrical figures using switch statement.

```c
#include <stdio.h>
main()
{
        int  fig_code;
        float  side, base, length, bredth, height, area, radius;
        printf("------------------------\n");
        printf(" 1 --> Circle\n");
        printf(" 2 --> Rectangle\n");
        printf(" 3 --> Triangle\n");
        printf(" 4 --> Square\n");
        printf("------------------------\n");

        printf("Enter the Figure code\n");
        scanf("%d", &fig_code);

        switch(fig_code)
        {
                case 1:
                        printf(" Enter the radius\n");
                        scanf("%f",&radius);
                        area=3.142*radius*radius;
                        printf("Area of a circle=%f\n", area);
                        break;
                case 2:
                        printf(" Enter the bredth and length\n");
                        scanf("%f %f",&bredth, &length);
                        area=bredth *length;
                        printf("Area of a Reactangle=%f\n", area);
                        break;
                case 3:
                        printf(" Enter the base and height\n");
                        scanf("%f %f", &base, &height);
                        area=0.5 *base*height;
                        printf("Area of a Triangle=%f\n", area);
                        break;
                case 4:
                        printf(" Enter the side\n");
                        scanf("%f", &side);
                        area=side * side;
                        printf("Area of a Square=%f\n", area);
                        break;
                default:
                        printf(" Error in figure code\n");
                        break;
        }

}
```

14.   Program to find the factorial of a number.

```c
#include <stdio.h>
main()
{
        int  i,fact=1,num;
        printf("Enter the number\n");
        scanf("%d",&num);
        if( num <0)
                printf("Factorial is not there for –ve numbers");
        else if(num==0 || num==1)
                fact=1;
        else
        {
                for(i=1;i<=num; i++)
                        fact *= i;
        }
        printf(" Factorial of %d =%5d\n", num,fact);
}
```

15. Program to illustrate for loop without initial and increment/decrement expressions.

```c
#include <stdio.h>
main()
{
        int  i=0,limit=5;
        printf(" Values of I\n");
        for(  ; i<limit;   )
        {
                i++;
                printf("%d\n", i);
        }
}
```

16.   Program to accept a string and find the sum of all digits in the string.

```c
#include <stdio.h>
main()
{
        char string[80];
        int count, nc=0, sum=0;
        printf("Enter the string containing both digits and alphabet\n");
        scanf("%s", string);
        for(count=0; string[count]!='\0'; count++)
        {
                if((string[count]>='0') && (string[count]<='9'))
                {
                        nc += 1;
                        sum += (string[count] - '0');
                }
        }
        printf("NO. of Digits in the string= %d\n",nc);
        printf("Sum of all digits= %d\n",sum);

}
```

17.   Program to find the sum of the sine series.

```c
#include <stdio.h>
#include <math.h>
#define pi 3.142

main()
{
        int  i,n,k,sign;
        float sum=0,num,den,xdeg,xrad,xsqr,term;
        printf("Enter the angle( in degree): \n");
        scanf("%f",&xdeg);
        printf("Enter the no. of terms: \n");
        scanf("%d",&n);
        xrad=xdeg * (pi/180.0);    /* Degrees to radians*/
        xsqr= xrad*xrad;
        sign=1; k=2; num=xrad; den=1;

        for(i=1;i<=n; i++)
        {
                term=(num/den)* sign;
                sum += term;
                sign *= -1;
                num *= xsqr;
                den *= k*(k+1);
                k += 2;
        }
        printf("Sum of sine series of %d terms =%8.3f\n",n,sum);
}
```

18.   Program to find the sum of cos(x) series.

```c
#include<stdio.h>
#include<math.h>
main()
{
        float x, sign, cosx, fact;
        int n,x1,i,j;
        printf("Enter the number of the terms in a series\n");
        scanf("%d", &n);
        printf("Enter the value of x(in degrees)\n");
        scanf("%f", &x);
        x1=x;
        x=x*(3.142/180.0); /* Degrees to radians*/
        cosx=1;
        sign=-1;
        for(i=2; i<=n; i=i+2)
        {
                fact=1;
                for(j=1;j<=i;j++)
                {
                        fact=fact*j;
                }
                cosx=cosx+(pow(x,i)/fact)*sign;
                sign=sign*(-1);
        }
```

```c
        printf("Sum of the cosine series=%f\n", cosx);
        printf("The value of cos(%d) using library function=%f\n",x1,cos(x));
}
```

19.   Program to reverse the given integer.

```c
#include <stdio.h>

main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);
}
```

20.   Program to accept a decimal number and convert to binary and count the number of 1's in the binary number.

```c
#include <stdio.h>
main()
{
        long   num, dnum, bin = 0, base = 1;
        int rem, no_of_1s = 0 ;
        printf("Enter a decimal integer:\n");
        scanf("%ld", &num);                             /*maximum five digits */
        dnum = num;
        while( num > 0)
        {
                rem = num % 2;
                if (rem==1)                     /*To count number of 1s*/
                {
                        no_of_1s++;
                }
                bin = bin + rem * base;
                num = num / 2 ;
                base = base * 10;
        }
        printf("Input number is = %ld\n", dnum);
        printf("Its Binary equivalent is =%ld\n", bin);
        printf("No. of 1's in the binary number is = %d\n", no_of_1s);
}
```

Output:

Enter a decimal integer: 75
Input number is = 75
Its Binary equivalent is =1001011
No. of 1's in the binary number is = 4

RUN2
Enter a decimal integer: 128
Input number is = 128
Its Binary equivalent is=10000000
No. of 1's in the binary number is = 1

21.   Program to find the number of characters, words and lines.

```c
#include<conio.h>
#include<string.h>
#include<stdio.h>
void main()
{
       int count=0,chars,words=0,lines,i;
       char text[1000];
       clrscr();
       puts("Enter text:");
       gets(text);
       while (text[count]!='\0')
              count++;
       chars=count;
       for (i=0;i<=count;i++)
       {
              if ((text[i]==' '&&text[i+1]!=' ')||text[i]=='\0')
                words++;
       }
       lines=chars/80+1;
       printf("no. of characters: %d\n", chars);
       printf("no. of words: %d\n", words);
       printf("no. of lines: %d\n", lines);
       getch();
}
```


22.   Program to find the GCD and LCM of two integers output the results along with the given integers. Use Euclids' algorithm.

```c
#include <stdio.h>
main()
{
       int  num1, num2, gcd, lcm, remainder, numerator, denominator;
       clrscr();
       printf("Enter two numbers: \n");
       scanf("%d %d", &num1,&num2);
       if (num1 > num2)
       {
              numerator = num1;
              denominator = num2;
       }
```

```
        else
        {
                numerator = num2;
                denominator = num1;
        }
        remainder = numerator % denominator;
        while(remainder !=0)
        {
                numerator   = denominator;
                denominator = remainder;
                remainder   = numerator % denominator;
        }
        gcd = denominator;
        lcm = num1 * num2 / gcd;
        printf("GCD of %d and %d = %d \n", num1,num2,gcd);
        printf("LCM of %d and %d = %d \n", num1,num2,lcm);
}
```

Output:

Enter two numbers: 5 15
GCD of 5 and 15 = 5
LCM of 5 and 15 = 15

23.   Program to find the sum of odd numbers and sum of even numbers from 1 to
N. Output the computed sums on two different lines with suitable headings.

```
#include <stdio.h>
main()
{
            int i, N, oddsum = 0, evensum = 0;
            printf("Enter the value of N: \n");
            scanf ("%d", &N);
            for (i=1; i <=N; i++)
            {
                        if (i % 2 == 0)
                                evensum = evensum + i;
                        else
                                oddsum = oddsum + i;
            }
            printf ("Sum of all odd numbers  = %d\n", oddsum);
            printf ("Sum of all even numbers = %d\n", evensum);
}
```

Output:

RUN1
Enter the value of N: 10
Sum of all odd numbers = 25
Sum of all even numbers = 30

RUN2
Enter the value of N: 50
Sum of all odd numbers = 625
Sum of all even numbers = 650

24.   Program to check whether a given number is prime or not and output the given number with suitable message.

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
        int num, j, flag;
        clrscr();

        printf("Enter a number: \n");
        scanf("%d", &num);
        if ( num <= 1)
        {
                printf("%d is not a prime numbers\n", num);
                exit(1);
        }
        flag = 0;
        for ( j=2; j<= num/2; j++)
        {
                if( ( num % j ) == 0)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 0)
                printf("%d is a prime number\n",num);
        else
                printf("%d is not a prime number\n", num);
}
```

Output:

RUN 1
Enter a number: 34
34 is not a prime number

RUN 2
Enter a number: 29
29 is a prime number

25.   Program to generate and print prime numbers in a given range. Also print the number of prime numbers.

```c
#include <stdio.h>
#include <math.h>
main()
{
        int M, N, i, j, flag, temp, count = 0;
        clrscr();
        printf("Enter the value of M and N: \n");
        scanf("%d %d", &M,&N);
        if(N < 2)
        {
                printf("There are no primes upto %d\n", N);
```

```
                exit(0);
        }
        printf("Prime numbers are\n");
        temp = M;
        if ( M % 2 == 0)
        {
                M++;
        }
        for (i=M; i<=N; i=i+2)
        {
                flag = 0;
                for (j=2; j<=i/2; j++)
                {
                        if( (i%j) == 0)
                        {
                                flag = 1;
                                break;
                        }
                }
                if(flag == 0)
                {
                        printf("%d\n",i);
                        count++;
                }
        }
        printf("Number of primes between %d  and %d = %d\n",temp,N,count);
}
```

Output:

Enter the value of M and N: 15 45
Prime numbers are
17
19
23
29
31
37
41
43
Number of primes between 15 and 45 = 8

26.   Write to accept a 1-dimensional array of N elements & split into 2 halves & sort
        1st half in ascending order & 2nd into descending order.

```
#include<stdio.h>
main ()
{
        int i,j,a,n,b,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        b = n/2;
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<b; ++i)
```

```c
        {
                for (j=i+1; j<b; ++j)
                {
                        if (number[i] > number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        for (i=b; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] < number[j])
                                {
                                        a = number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
         }
        printf (" The 1st half numbers\n");
        printf (" arranged in asc\n");
        for (i=0; i<b; ++i)
                printf ("%d ",number[i]);
        printf("\nThe 2nd half Numbers\n");
        printf("order arranged in desc.order\n");
        for(i=b;i<n;i++)
                printf("%d ",number[i]);
}
```

27.   Program to delete the desired element from the list.

```c
# include <stdio.h>
main()
{
        int  vectx[10];
        int  i, n, found = 0, pos, element;
        printf("Enter how many elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &vectx[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", vectx[i]);
        }
         printf("Enter the element to be deleted\n");
         scanf("%d",&element);
        for(i=0; i<n; i++)
        {
```

```c
                if ( vectx[i] == element)
                {
                        found = 1;
                        pos = i;
                        break;
                }
        }
        if (found == 1)
        {
                for(i=pos; i< n-1; i++)
                {
                        vectx[i] = vectx[i+1];
                }
                printf("The resultant vector is \n");
                for(i=0; i<n-1; i++)
                {
                        printf("%d\n",vectx[i]);
                }
        }
        else
                printf("Element %d is not found in the vector\n", element);
}
```

28.   Write a "C" program to Interchange the main diagonal elements with the
        scondary diagonal elements.

```c
#include<stdio.h>
main ()
{
        int i,j,m,n,a;
        static int ma[10][10];
        printf ("Enetr the order of the matix \n");
        scanf ("%dx%d",&m,&n);
        if (m==n)
        {
                printf ("Enter the co-efficients of the matrix\n");
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                scanf ("%d",&ma[i][j]);
                        }
                }
                printf ("The given matrix is \n");
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                printf (" %d",ma[i][j]);
                        }
                        printf ("\n");
                }
                for (i=0;i<m;++i)
                {
                        a = ma[i][i];
                        ma[i][i]   = ma[i][m-i-1];
```

```
                ma[i][m-i-1] = a;
        }
        printf ("THe matrix after changing the \n");
        printf ("main diagonal & secondary diagonal\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
}
else
        printf ("The given order is not square matrix\n");
}
```

29   Program to insert an element at an appropriate position in an array.

```
#include <stdio.h>
#include <conio.h>
main()
{
        int  x[10];
        int  i, j, n, m, temp, key, pos;
        clrscr();
        printf("Enter how many elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &x[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }
        for(i=0; i< n; i++)
        {
                for(j=i+1; j<n; j++)
                {
                        if (x[i] > x[j])
                        {
                                temp = x[i];
                                x[i] = x[j];
                                x[j] = temp;
                        }
                }
        }
        printf("Sorted list is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }
        printf("Enter the element to be inserted\n");
```

```c
        scanf("%d",&key);
        for(i=0; i<n; i++)
        {
                if ( key < x[i] )
                {
                        pos = i;
                        break;
                }
        }
        m = n - pos + 1 ;
        for(i=0; i<= m ; i++)
        {
                x[n-i+2] = x[n-i+1] ;
        }
        x[pos] = key;

        printf("Final list is:\n");
        for(i=0; i<n+1; i++)
        {
                printf("%d\n", x[i]);
        }
}
```

30.   Program to compute mean, varience and standard deviation.

```c
main()
{
                float x[10];
                int  i, n;
                float avrg, var, SD, sum=0, sum1=0;

                printf("Enter how many elements\n");
                scanf("%d", &n);
                printf("Enter %d numbers:",n);
                for(i=0; i<n; i++)
                {
                        scanf("%f", &x[i]);
                }
                                /* Compute the sum of all elements */
                for(i=0; i<n; i++)
                        sum = sum + x[i];

                avrg = sum /(float) n;
                                /* Compute varaience and standard deviation  */
                for(i=0; i<n; i++)
                {
                        sum1 = sum1 + pow((x[i] - avrg),2);
                }
                var = sum1 / (float) n;
                SD = sqrt(var);
                printf("Average of all elements =%.2f\n", avrg);
                printf("Variance of all elements =%.2f\n", avrg);
                printf("Standard Deviation of all elements =%.2f\n", avrg);
}
```

31. Program to find the frequency of odd numbers & even numbers in the input of a matrix.

```c
#include<stdio.h>
main ()
{
        int i,j,m,n,even=0,odd=0;
        static int ma[10][10];
        printf ("Enter the order ofthe matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefficients if matrix \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d", &ma[i][j]);
                        if ((ma[i][j]%2) == 0)
                        {
                                ++even;
                        }
                        else
                                ++odd;
                }
        }
        printf ("The given matrix is\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                        printf (" %d",ma[i][j]);
                printf ("\n");
        }
        printf ("The frequency of odd number occurrence = %d\n",odd);
        printf ("The frequency of even number occurrence = %d\n",even);
}
```

32. Program to sort all rows of matrix in ascending order & all columns in descending order.

```c
#include <stdio.h>
main ()
{
        int i,j,k,a,m,n;
        static int ma[10][10],mb[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d", &m,&n);
        printf ("Enter co-efficients of the matrix \n");

        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                        mb[i][j] = ma[i][j];
                }
        }
        printf ("The given matrix is \n");
```

```c
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
                printf (" %d",ma[i][j]);
        }
        printf ("\n");
}
printf ("After arranging rows in ascending order\n");
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
                for (k=(j+1);k<n;++k)
                {
                        if (ma[i][j] > ma[i][k])
                        {
                                a = ma[i][j];
                                ma[i][j] = ma[i][k];
                                ma[i][k] = a;
                        }
                }
        }
}
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
                printf (" %d",ma[i][j]);
        }
        printf ("\n");
}
printf ("After arranging the columns in descending order \n");
for (j=0;j<n;++j)
{
        for (i=0;i<m;++i)
        {
                for (k=i+1;k<m;++k)
                {
                        if (mb[i][j] < mb[k][j])
                        {
                                a = mb[i][j];
                                mb[i][j] = mb[k][j];
                                mb[k][j] = a;
                        }
                }
        }
}
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
                printf (" %d",mb[i][j]);
        }
        printf ("\n");
}
}
```

33.   Program to convert lower case letters to upper case vice-versa.

```c
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
main()
{
        char sentence[100];
        int count, ch, i;
        clrscr();
        printf("Enter a sentence\n");
        for(i=0; (sentence[i] = getchar())!='\n'; i++);
        count = i;
        sentence[count]='\0';
        printf("Input sentence is : %s",sentence);
        printf("\nResultant sentence is\n");
        for(i=0; i < count; i++)
        {
                ch = islower(sentence[i]) ? toupper(sentence[i]) : tolower(sentence[i]);
                putchar(ch);
        }
}
```

34.   Program to find the sum of the rows & columns of a matrix.

```c
main ()
{
        int i,j,m,n,sum=0;
        static int m1[10][10];
        printf ("Enter the order of the matrix\n");
        scanf ("%d%d", &m,&n);
        printf ("Enter the co-efficients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&m1[i][j]);
                }
        }
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        sum = sum + m1[i][j] ;
                }
                printf ("   Sum of the %d row is = %d\n",i,sum);
                sum = 0;
        }
        sum=0;
        for (j=0;j<n;++j)
        {
                for (i=0;i<m;++i)
                {
                        sum = sum+m1[i][j];
                }
                 printf ("Sum of the %d column is = %d\n", j,sum);
```

```
                sum = 0;
        }
 }


35.     Program to find the transpose of a matrix.

#include<stdio.h>
main ()
{
        int i,j,m,n;
        static int ma[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefiicients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
        printf ("Transpose of matrix is \n");
        for (j=0;j<n;++j)
        {
                for (i=0;i<m;++i)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
 }
```

36.   Program to accepts two strings and compare them. Finally print whether, both
are equal, or first string is greater than the second or the first string is less than the
second string without using string library.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int count1=0,count2=0,flag=0,i;
        char str1[10],str2[10];
        clrscr();
        puts("Enter a string:");
        gets(str1);
        puts("Enter another string:");
```

```
        gets(str2);
                                        /*Count the number of characters in str1*/
        while (str1[count1]!='\0')
              count1++;
                                        /*Count the number of characters in str2*/
        while (str2[count2]!='\0')
              count2++;
        i=0;

/*The string comparison starts with the first character in each string and continues with
subsequent characters until the corresponding characters differ or until the end of the
strings is reached.*/

        while ( (i < count1) && (i < count2))
        {
              if (str1[i] == str2[i])
              {
                     i++;
                     continue;
              }
              if (str1[i]<str2[i])
              {
                     flag = -1;
                     break;
              }
              if (str1[i] > str2[i])
              {
                     flag = 1;
                     break;
              }
        }
        if (flag==0)
              printf("Both strings are equal\n");
        if (flag==1)
              printf("String1 is greater than string2\n", str1, str2);
        if (flag == -1)
              printf("String1 is less than string2\n", str1, str2);
        getch();
}
```

Output:

Enter a string: happy
Enter another string: HAPPY
String1 is greater than string2

RUN2:
Enter a string: Hello
Enter another string: Hello
Both strings are equal

RUN3:
Enter a string: gold
Enter another string: silver
String1 is less than string2

37.   Program to accept N integer number and store them in an array AR. The odd elements in the AR are copied into OAR and other elements are copied into EAR. Display the contents of OAR and EAR.

```c
 #include <stdio.h>
main()
{
        long int ARR[10], OAR[10], EAR[10];
        int i,j=0,k=0,n;
        printf("Enter the size of array AR:\n");
        scanf("%d",&n);
        printf("Enter the elements of the array:\n");
        for(i=0;i<n;i++)
        {
                scanf("%ld",&ARR[i]);
                fflush(stdin);
        }
                        /*Copy odd and even elemets into their respective arrays*/
        for(i=0;i<n;i++)
        {
                if (ARR[i]%2 == 0)
                {
                        EAR[j] = ARR[i];
                        j++;
                }
                else
                {
                        OAR[k] = ARR[i];
                        k++;
                }
        }
        printf("The elements of OAR are:\n");
        for(i=0;i<j;i++)
        {
                printf("%ld\n",OAR[i]);
        }

        printf("The elements of EAR are:\n");
        for(i=0;i<k;i++)
        {
                printf("%ld\n", EAR[i]);
        }
}
```

 Output:

Enter the size of array AR: 6
Enter the elements of the array:
12
345
678
899
900
111

The elements of OAR are:
 345

```
 899
 111
```

The elements of EAR are:
```
 12
 678
 900
```


38.   Program to find the sub-string in a given string.

```c
#include<stdio.h>
#include<conio.h>
void main()
{
        int count1=0,count2=0,i,j,flag;
        char str[80],search[10];
        clrscr();
        puts("Enter a string:");
        gets(str);
        puts("Enter search substring:");
        gets (search);
        while (str[count1]!='\0')
                count1++;
        while (search[count2]!='\0')
                count2++;
        for(i=0;i<=count1-count2;i++)
        {
                for(j=i;j<i+count2;j++)
                {
                        flag=1;
                        if (str[j]!=search[j-i])
                        {
                                flag=0;
                                break;
                        }
                }
                if (flag==1)
                        break;
        }
        if (flag==1)
                puts("SEARCH SUCCESSFUL!");
        else
                puts("SEARCH UNSUCCESSFUL!");
        getch();
}
```


39.   Program to accept a string and find the number of times the word 'the' appears in it.

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

void main()
{
        int count=0,i,times=0,t,h,e,space;
```

```c
    char str[100];
    clrscr();
    puts("Enter a string:");
    gets(str);
                        /*Traverse the string to count the number of characters*/
    while (str[count]!='\0')
    {
            count++;
    }
                        /*Finding the frequency of the word 'the'*/
    for(i=0;i<=count-3;i++)
    {
            t=(str[i]=='t'||str[i]=='T');
            h=(str[i+1]=='h'||str[i+1]=='H');
            e=(str[i+2]=='e'||str[i+2]=='E');
            space=(str[i+3]==' '||str[i+3]=='\0');
            if ((t&&h&&e&&space)==1)
               times++;
    }
    printf("Frequency of the word \'the\' is %d\n",times);
    getch();
}
```

Output:

Enter a string: The Teacher's day is the birth day of Dr.S.Radhakrishnan
Frequency of the word 'the' is 2

40.   Program to find the length of a string without using the built-in function, also
check whether it is a palindrome or not.

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

main()
{
     char string[25], revString[25]={'\0'};
     int  i,length = 0, flag = 0;
     clrscr();
     fflush(stdin);
     printf("Enter a string\n");
     gets(string);
     for (i=0; string[i] != '\0'; i++) /*keep going through each */
     {                          /*character of the string */
           length++;                 /*till its end */
     }
     printf("The length of the string: \'%s\' = %d\n", string, length);

     for (i=length-1; i >= 0 ; i--)
     {
       revString[length-i-1] = string[i];
     }
                    /*Compare the input string and its reverse. If both are equal
                         then the input string is palindrome. Otherwise it is
                                not a palindrome */
     for (i=0; i < length ; i++)
```

```
            {
                if (revString[i] == string[i])
                        flag = 1;
                else
                        flag = 0;
            }
        if (flag == 1)
                printf ("%s is a palindrome\n", string);
        else
                printf("%s is not a palindrome\n", string);
 }
```

Output:

Enter a string: madam
The length of the string 'madam' = 5
madam is a palindrome

RUN2:
Enter a string: good
The length of the string 'good' = 4
good is not a palindrome

41.   Program to accept two strings and concatenate them i.e. The second string is appended to the end of the first string.

```
#include <stdio.h>
#include <string.h>
main()
 {
    char string1[20], string2[20];
    int i,j,pos;
    strset(string1, '\0');              /*set all occurrences in two strings to NULL*/
    strset(string2,'\0');
    printf("Enter the first string:");
    gets(string1);
    fflush(stdin);
    printf("Enter the second string:");
    gets(string2);
    printf("First string  = %s\n", string1);
    printf("Second string = %s\n", string2);
                            /*To concate the second stribg to the end of the string
     travserse the first to its end and attach the second string*/
    for (i=0; string1[i] != '\0'; i++)
    {
        ;                       /*null statement: simply trvsering the string1*/
    }
    pos = i;
    for (i=pos,j=0; string2[j]!='\0'; i++)
    {
        string1[i] = string2[j++];
    }
    string1[i]='\0';            /*set the last character of string1 to NULL*/
    printf("Concatenated string = %s\n", string1);
}
```

Output:

Enter the first string: CD-

Enter the second string: ROM

First string = CD-
Second string = ROM

Concatenated string = CD-ROM

42.   Program for Comparing two matrices for equality.

```c
#include <stdio.h>
#include <conio.h>
main()
{
        int A[10][10], B[10][10];
        int i, j, R1, C1, R2, C2, flag =1;
        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);
        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        printf("Enter the elements of matrix A\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        printf("Enter the elements of matrix B\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
        printf("MATRIX A is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
        printf("MATRIX B is\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
```

```
        }
                            /* Comparing two matrices for equality */
        if(R1 == R2 && C1 == C2)
        {
                printf("Matrices can be compared\n");
                for(i=0; i<R1; i++)
                {
                        for(j=0; j<C2; j++)
                        {
                                if(A[i][j] != B[i][j])
                                {
                                        flag = 0;
                                        break;
                                }
                        }
                }
        }
        else
        {
                printf(" Cannot be compared\n");
                exit(1);
        }
        if(flag == 1 )
                printf("Two matrices are equal\n");
        else
                printf("But,two matrices are not equal\n");
}
```

43.   Program to illustrate how user authentication is made before allowing the user
to access the secured resources. It asks for the user name and then the password. The
password that you enter will not be displayed, instead that character is replaced by '*'.

```
#include <stdio.h>
void main()
{
        char pasword[10],usrname[10], ch;
        int i;
        clrscr();
        printf("Enter User name: ");
        gets(usrname);
        printf("Enter the password <any 8 characters>: ");
        for(i=0;i<8;i++)
        {
                ch = getch();
                pasword[i] = ch;
                ch = '*' ;
                printf("%c",ch);
        }
        pasword[i] = '\0';
        printf("\n\n\nYour password is :");
        for(i=0;i<8;i++)
        {
                 printf("%c",pasword[i]);
        }
}
```

44.   Program to find the length of a string without using the built-in function.

```c
# include <stdio.h>

main()
{
      char string[50];
      int  i, length = 0;
      printf("Enter a string\n");
      gets(string);
      for (i=0; string[i] != '\0'; i++) /*keep going through each */
      {                               /*character of the string */
         length++;                    /*till its end */
      }
      printf("The length of a string is the number of characters in it\n");
      printf("So, the length of %s =%d\n", string, length);
}
```

Output:

Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5

RUN2
Enter a string
E-Commerce is hot now
The length of a string is the number of characters in it
So, the length of E-Commerce is hot now =21

45.   Program to read N integers (zero, +ve and -ve) into an array A and to
       a) Find the sum of negative numbers
       b) Find the sum of positive numbers and
       c) Find the average of all input numbers

Output the various results computed with proper headings.

```c
#include <stdio.h>
#define MAXSIZE 10
main()
{
      int array[MAXSIZE];
      int i, N, negsum=0, posum=0, count1=0, count2=0;
      float total=0.0, averg;
      clrscr();
      printf ("Enter the value of N\n");
      scanf("%d", &N);
      printf("Enter %d numbers (-ve, +ve and zero)\n", N);
      for(i=0; i< N ; i++)
      {
            scanf("%d",&array[i]);
            fflush(stdin);
      }
      printf("Input array elements\n");
```

```c
        for(i=0; i< N ; i++)
        {
                printf("%+3d\n",array[i]);
        }
                    /* Summing  begins */
        for(i=0; i< N ; i++)
        {
                if(array[i] < 0)
                {
                        negsum = negsum +  array[i];
                }
                else if(array[i] > 0)
                {
                        posum = posum + array[i];
                }
                else if( array[i] == 0)
                {
                            ;
                }
                total = total + array[i] ;
        }
        averg = total / N;
        printf("\nSum of all negative numbers    = %d\n",negsum);
        printf("Sum of all positive numbers    = %d\n", posum);
        printf("\nAverage of all input numbers   = %.2f\n", averg);
}
```

Output:

```
Enter the value of N: 5
Enter 5 numbers (-ve, +ve and zero)
5
-3
0
-7
6

Input array elements
 +5
 -3
 +0
 -7
 +6

Sum of all negative numbers    = -10
Sum of all positive numbers   = 11
Average of all input numbers   = 0.20
```

46.   Program to sort N numbers in ascending order using Bubble sort and print both the given and the sorted array with suitable headings.

```c
#include <stdio.h>
#define MAXSIZE 10
main()
{
        int array[MAXSIZE];
```

```c
        int i, j, N, temp;
        clrscr();
        printf("Enter the value of N: \n");
        scanf("%d",&N);
        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array is\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
                        /* Bubble sorting begins */
        for(i=0; i< N ; i++)
        {
                for(j=0; j< (N-i-1) ; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                temp      = array[j];
                                array[j]   = array[j+1];
                                array[j+1] = temp;
                        }
                }
        }
        printf("Sorted array is...\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
}
```

Output:

Enter the value of N: 5
Enter the elements one by one
390
234
111
876
345

Input array is
390
234
111
876
345

Sorted array is...
111
234
345
390
876

47.   Program to accept N numbers sorted in ascending order and to search for a given number using binary search. Report success or failure in the form of suitable messages.

```c
#include <stdio.h>
main()
{
        int array[10];
        int i, j, N, temp, keynum, ascending = 0;
        int low,mid,high;
        clrscr();
        printf("Enter the value of N: \n");
        scanf("%d",&N);
        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array elements\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
                                /* Bubble sorting begins */
        for(i=0; i< N ; i++)
        {
                for(j=0; j< (N-i-1) ; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                temp = array[j];
                                array[j] = array[j+1];
                                array[j+1] = temp;
                        }
                }
        }
        printf("Sorted array is...\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
        printf("Enter the element to be searched\n");
        scanf("%d", &keynum);
                        /* Binary searching begins */
        low=1;
        high=N;
        do
        {
                mid= (low + high) / 2;
                if ( keynum < array[mid] )
                        high = mid - 1;
                else if ( keynum > array[mid])
                        low = mid + 1;
        } while( keynum!=array[mid] && low <= high);         /* End of do- while */
        if( keynum == array[mid] )
        {
                printf("SUCCESSFUL SEARCH\n");
```

```
		}
		else
			printf("Search is FAILED\n");
}
```

Output:

```
Enter the value of N: 4
Enter the elements one by one
3
1
4
2


Input array elements
3
1
4
2

Sorted array is...
1
2
3
4

Enter the element to be searched
4

SUCCESSFUL SEARCH
```

48.   Program read a sentence and count the number of number of vowels and consonants in the given sentence. Output the results on two lines with suitable headings.

```
#include <stdio.h>

main()
{
	char sentence[80];
	int i, vowels=0, consonants=0, special = 0;
	clrscr();
	printf("Enter a sentence\n");
	gets(sentence);
	for(i=0; sentence[i] != '\0'; i++)
	{
		if((sentence[i] == 'a'||sentence[i] == 'e'||sentence[i] == 'i'||
		sentence[i] == 'o'||sentence[i] == 'u') ||(sentence[i] == 'A'||
		sentence[i] == 'E'||sentence[i] == 'I'|| sentence[i] == 'O'||
		sentence[i] == 'U'))
		{
			vowels = vowels + 1;
		}
		else
		{
```

```
                    consonants = consonants + 1;
            }
             if (sentence[i] =='\t' ||sentence[i] =='\0' || sentence[i] ==' ')
            {
                    special = special + 1;
            }
        }
        consonants = consonants - special;
        printf("No. of vowels in %s = %d\n", sentence, vowels);
        printf("No. of consonants in %s = %d\n", sentence, consonants);
}
```

Output:

Enter a sentence
Good Morning
No. of vowels in Good Morning = 4
No. of consonants in Good Morning = 7

49.   Program to sort names in alphabetical order using structures.

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

struct tag
{
        char name[10];
        int rno;
};

typedef struct tag node;

node  s[5];

sort(int no)
{
        int i,j;
        node temp;
        for(i=0;i<no-1;i++)
                for(j=i+1;j<no;j++)
                        if(strcmp(s[i].name,s[j].name)>0)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
}

main()
{
        int no,i;
        clrscr();
        fflush(stdin);
        printf("Enter The Number Of Students:");
        scanf("%d",&no);
```

```c
        for(i=0;i<no;i++)
        {
                printf("Enter The Name:");
                fflush(stdin);
                gets(s[i].name);
                printf("Enter the Roll:");
                scanf("%d",&s[i].rno);
        }
        sort(no);
        for(i=0;i<no;i++)
        {
                printf("%s\t",s[i].name);
                printf("%d\n",s[i].rno);
        }
        getche();
}
```

50.   Program to illustrate the unions.

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

main()
{
        union number
        {
                int  n1;
                float n2;
        };
        union number x;
        clrscr() ;
        printf("Enter the value of n1: ");
        scanf("%d", &x.n1);
        printf("Value of n1 =%d", x.n1);
        printf("\nEnter the value of n2: ");
        scanf("%d", &x.n2);
        printf("Value of n2 = %d\n",x.n2);
}
```

51.   Program to store the inventory information system.

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

void main()
{      struct date
        {
                int day;
                int month;
                int year;
        };
        struct details
        {
                char name[20];
```

```c
                int price;
                int code;
                int qty;
                struct date mfg;
        };

        struct details item[50];
        int n,i;
        clrscr();
        printf("Enter number of items:");
        scanf("%d",&n);
        fflush(stdin);
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                printf("Item name:");
                scanf("%s",item[i].name);
                fflush(stdin);
                printf("Item code:");
                scanf("%d",&item[i].code);
                fflush(stdin);
                printf("Quantity:");
                scanf("%d",&item[i].qty);
                fflush(stdin);
                printf("price:");
                scanf("%d",&item[i].price);
                fflush(stdin);
                printf("Manufacturing date(dd-mm-yyyy):");
                scanf("%d-%d-%d",&item[i].mfg.day,&item[i].mfg.month,&item[i].mfg.year);
        }
        printf("          *****  INVENTORY *****\n");
        printf("-------------------------------------------------------------\n");
        printf("S.N.|   NAME          |  CODE  | QUANTITY | PRICE |MFG.DATE\n");
        printf("-------------------------------------------------------------\n");
        for(i=0;i<n;i++)
        printf("%d    %-15s       %-d         %-5d    %-5d
                %d/%d/%d\n",i+1,item[i].name,item[i].code,item[i].qty,item[i].price,ite
                m[i].mfg.day,item[i].mfg.month,item[i].mfg.year);
        printf("-------------------------------------------------------------\n");
        getch();
}
```

52.   Program to find the sum and difference of two matrices.

```c
#include <stdio.h>

 int A[10][10], B[10][10], sumat[10][10], diffmat[10][10];
 int i, j, R1, C1, R2, C2;

main()
{
        void readmatA();
        void printmatA();
        void readmatB();
        void printmatB();
        void sum();
```

```c
        void diff();
        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);
        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        if( R1 != R2 && C1 != C2)
        {
                printf("Addition and subtraction are possible\n");
                exit(1);
        }
        else
        {
                printf("Enter the elements of matrix A\n");
                readmatA();
                printf("MATRIX A is\n");
                printmatA();
                printf("Enter the elements of matrix B\n");
                readmatB();
                printf("MATRIX B is\n");
                printmatB();
                sum();
                diff();
        }
        return 0;
}
void readmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        return;
}
void readmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
}
void printmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
}
```

```c
void printmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
}

void sum()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        sumat[i][j] = A[i][j] + B[i][j];
                }
        }
        printf("Sum matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",sumat[i][j]) ;
                }
                printf("\n");
        }
        return;
}

void diff()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        diffmat[i][j] = A[i][j] - B[i][j];
                }
        }
        printf("Difference matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",diffmat[i][j]);
                }
                printf("\n");
        }
        return;
}
```

53.   Program to read a matrix A (MxN) and to find the following using functions:
a) Sum of the elements of each row
b) Sum of the elements of each column
c) Find the sum of all the elements of the matrix

Output the computed results with suitable headings

```c
#include <stdio.h>
main()
{
        int arr[10][10];
        int i, j, row, col, rowsum, colsum,sumall=0;
        clrscr();

        printf("Enter the order of the matrix\n");
        scanf("%d %d", &row, &col);

        printf("Enter the elements of the matrix\n");
        for(i=0; i<row; i++)
        {
                for(j=0; j< col; j++)
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("Input matrix is\n");
        for(i=0; i<row; i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("%3d", arr[i][j]);
                }
                printf("\n");
        }
                            /* computing row sum */
        for(i=0; i<row; i++)
        {
                rowsum = Addrow(arr,i,col);
                printf("Sum of row %d = %d\n", i+1, rowsum);
        }

        for(j=0; j<col; j++)
        {
                colsum = Addcol(arr,j,row);
                printf("Sum of column  %d = %d\n", j+1, colsum);
        }
                            /* computation of all elements */
        for(j=0; j< row; j++)
        {
                sumall = sumall + Addrow(arr,j,col);
        }
        printf("Sum of all elements of matrix = %d\n", sumall);
}

/* Function to add each row */

int Addrow(int A[10][10], int k, int c)
```

```
{
        int rsum=0, i;
        for(i=0; i< c; i++)
        {
                rsum = rsum + A[k][i];
        }
        return(rsum);
}
```

/* Function to add each column */

```
int Addcol(int A[10][10], int k, int r)
{
        int csum=0, j;
        for(j=0; j< r; j++)
        {
                csum = csum + A[j][k];
        }
         return(csum);
}
```

Output:

Enter the order of the matrix
3
3

Enter the elements of the matrix
1 2 3
4 5 6
7 8 9

Input matrix is
  1 2 3
  4 5 6
  7 8 9
Sum of row 1 = 6
Sum of row 2 = 15
Sum of row 3 = 24
Sum of column 1 = 12
Sum of column 2 = 15
Sum of column 3 = 18
Sum of all elements of matrix = 45

54.   Program to find the sum of all elements of an array using pointers.

#include <stdio.h>

```
main()
{
        static int array[5]={ 200,400,600,800,1000 };
        int addnum(int *ptr);                          /* function prototype */

        int sum;
        sum = addnum(array);
        printf(" Sum of all array elements=%d\n", sum);
```

```
}

int  addnum(int *ptr)
{
        int total=0, index;
        for(index=0;index<5; index++)
                total+=*(ptr + index);
        return(total);
}
```

55.   Program to swap particular elements in array using pointers.

```
#include <stdio.h>

main()
{
        float x[10];
        int i,n;
        float swap34(float *ptr1, float  *ptr2 );     /* Function Declaration */
        printf(" How many Elements...\n");
        scanf("%d", &n);
        printf(" Enter Elements one by one\n");
        for(i=0;i<n;i++)
                scanf("%f",x+i);
        swap34(x+2, x+3);                              /* Interchanging 3rd element by 4th
*/
        printf(" Resultant Array...\n");
        for(i=0;i<n;i++)
                printf("X[%d]=%f\n",i,x[i]);
}

float swap34(float *ptr1, float *ptr2 )             /* Function Definition */
{
        float temp;
        temp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=temp;
}
```

56.   Program to find the sum of two 1-D arrays using Dynamic Memory Allocation.

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

main()
{
        int i,n,sum;
        int *a,*b,*c;
        printf(" How many Elements in each array...\n");
        scanf("%d", &n);
        a=(int *) malloc(sizeof(n*(int));
        b=(int *) malloc(sizeof(n*(int));
        c=(int *) malloc(sizeof(n*(int));
        printf(" Enter Elements of First List\n");
        for(i=0;i<n;i++)
```

```
        scanf("%f",a+i);

printf(" Enter Elements of Second List\n");
for(i=0;i<n;i++)
        scanf("%f",b+i);

for(i=0;i<n;i++)
        c+i=(a+i) +( b+i);
printf(" Resultant List is\n");
for(i=0;i<n;i++)
        printf("C[%d]=%f\n",i,c[i]);
}
```

Chapter
3

Storage Classes, Functions and User Defined Data Types

3.1. Storage Classes:

Storage classes are used to define the scope (visibility) and life-time of variables and/or functions. Every variable and function in C has two attributes: type and storage class. There are four storage classes:

- auto
- static
- extern or global and
- register

3.1.1   The auto storage class:

Variables declared within function bodies are automatic by default. If a compound statement starts with variable declarations, then these variables can be acted on within the scope of the enclosing compound statement. A compound statement with declarations is called a block to distinguish it from one that does not begin with declarations.

Declarations of variables within blocks are implicitly of storage class automatic. The key-word auto can be used to explicitly specify the storage class. An example is:

```
auto int a, b, c;
auto float f;
```

Because the storage class is auto by default, the key-word auto is seldom used.

When a block is entered, the system allocates memory for the automatic variables. With in that block, these variables are defined and are considered "local" to the block. When the block is exited, the system releases the memory that was set aside for the automatic variables. Thus, the values of these variables are lost. If the block is reentered, the system once again allocates memory, but previous values are unknown. The body of a function definition constitutes a block if it contains declarations. If it does, then each invocation of the function set up a new environment.

3.1.2.   The static storage class:

It allows a local variable to retain its previous value when the block is reentered. This is in contrast to automatic variables, which lose their value upon block exit and must be reinitialized. The static variables hold their values throughout the execution of the program. As an example of the value-retention use of static, the outline of a function that behaves differently depending on how many times it has been called is:

```
void fun (void)
{
        static int cnt = 0
        ++ cnt;
        if (cnt % 2 == 0)
                . . .                                  /* do something */
        else
                . . .                                  /* do something different */
}
```

The first time the function is invoked, the variable cnt is initialized to zero. On function exit, the value of cnt is preserved in memory. Whenever the function is invoked again, cnt is not reinitialized. Instead, it retains its previous value from the last time the function was called. The declaration of cnt as a *static int* inside of fun () keeps it private of fun (). If it was declared outside of the function, then other functions could access it, too.

Static variables are classified into two types.

1.   internal static variables
2.   external static variables

External static variables are declared outside of all functions including the main() function. They are global variables but are declared with the keyword static. The external static variables cannot be used across multi-file program.

A static variable is stored at a fixed memory location in the computer, and is created and initialised once when the program is first started. Such a variable maintains its value between calls to the block (a function, or compound statement) in which it is defined. When a static variable is not initialized, it takes a value of zero.

3.1.3.   The storage class extern:

One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is extern. Such a variable is considered to be global to all functions declared after it, and upon exit from the block or function, the external variable remains in existence. The following program illustrates this:

```
# include <stdio.h>

int a = 1, b = 2, c = 3;                    /* global variable*/

int fun (void);                             /* function prototype*/

int main (void)
{
        printf ("%3d\n", fun ());           /* 12 is printed */
        printf ("%3d%3d%3d\n", a, b, c);    /* 4   2  3 is printed */
}

int fun (void)
{
        int b, c;
        a = b = c = 4;                      /* b and c are local */
        return (a + b + c);                 /* global b, c are masked*/
}
```

Note that we could have written:

    extern int a = 1, b = 2, c = 3;

The extern variables cannot be initialised in other functions whereas we can use for assignment.

This use of extern will cause some traditional C compilers to complain. In ANSI C, this use is allowed but not required. Variables defined outside a function have external storage class, even if the keyword extern is not used. Such variables cannot have auto or register storage class.

The keyword extern is used to tell the compiler to "look for it else where, either in this file or in some other file". Let us rewrite the last program to illustrate a typical use of the keyword extern:

In file file1.c

```
#include <stdio.h>
# include "file2.c"

int a=1, b=2, c=3;            /* external variable */

int fun (void)

int main (void)
{
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        printf ("The sum of a + b + c is: %3d\n", fun ());
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        return 0;
}
```

In file file2.c

```
int fun (void)
{
        extern int a;                  /* look for it elsewhere */
        int b, c;
        a = b = c = 4;
        return (a + b + c);
}
```

Output:

The values of a, b and c are: 1   2   3
The sum of a + b + c is: 12
The values of a, b and c are: 4   2   3

The two files can be compiled separately. The use of extern in the second file tells the compiler that the variable a will be defined elsewhere, either in this file or in some other. The ability to compile files separately is important when writing large programs.

External variables never disappear. Because they exist throughout the execution life of the program, they can be used to transmit values across functions. They may, however, be hidden if the identifier is redefined.

Information can be passed into a function two ways; by use of external variable and by use of the parameter passing mechanism. The use of the parameter mechanism is the best preferred method. Don't overuse 'extern' variables. It is usually better to pass lots of arguments to functions rather than to rely on hidden variables being passed (since you end up with clearer code and reusable functions).

3.1.4. The register storage class:

This is like `auto' except that it asks the compiler to store the variable in one of the CPU's fast internal registers. In practice, it is usually best not to use the `register' type since compilers are now so smart that they can do a better job of deciding which variables to place in fast storage than you can.

The use of storage class register is an attempt to improve execution speed. When speed is a concern, the programmer may choose a few variables that are most frequently accessed and declare them to be of storage class register. Common candidates for such treatment include loop variables and function parameters. The keyword register is used to declare these variables.

Examples:

1) register int x;
2) register int counter;

The declaration *register i*; is equivalent to *register int i;*

A complete summarized table to represent the lifetime and visibility (scope) of all the storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local

3.2. The type qualifiers, const and volatile:

The type qualifiers const and volatile restrict or qualify the way an identifier of a given type can be used.

3.2.1. Const Storage Class:

Variables can be qualified as 'const' to indicate that they are really constants, that can be initialised, but not altered. The declaration is:

const int k = 5;

As the type of k has been qualified by const, we can initialise k, but thereafter k cannot be assigned to, incremented, decremented or otherwise modified.

3.2.2. Volatile Storage Class:

Variables can also be termed 'volatile' to indicate that their value may change unexpectedly during the execution of the program. The declaration is:

extern const volatile int real_time_clock;

The extern means look for it elsewhere, either in this file or in some other file. The qualifier volatile indicates that the object may be acted on by the hardware. Because const is also a qualifier, the object may not be assigned to, incremented, or decremented within the program. The hardware can change the clock, but the code cannot.

3.3. FUNCTIONS:

Functions are a group of statements that have been given a name. This allows you to break your program down into manageable pieces and reuse your code.

The advantages of functions are:

1. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

2. The length of source program can be reduced by using function by using it at different places in the program according to the user's requirement.

3. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

4. They also allow more than one person to work on one program.

5. Function increases the execution speed of the program and makes the programming simple.

6. By using the function, portability of the program is very easy.

7. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

8. Debugging (removing error) becomes very easier and fast using the function sub-programming.

9. Functions are more flexible than library functions.

10. Testing (verification and validation) is very easy by using functions.

11. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

The functions are classified into standard functions and user-defined functions.

The standard functions are also called library functions or built in functions. All standard functions, such as sqrt(), abs(), log(), sin(), pow() etc. are provided in the library of functions. These are selective.

Most of the applications need other functions than those available in the software. These functions must be written by the user (programmer), hence, the name user-defined functions.

You can write as many functions as you like in a program as long as there is only one main (). As long as these functions are saved in the same file, you do not need to include a header file. They will be found automatically by the compiler.

There are 3 parts for using functions:

- Calling them,
- Defining them and
- Declaring them (prototypes).

3.3.1.   Calling functions:

When you wish to use a function, you can "call" it by name. Control is sent to the block of code with that name. Any values that the function needs to perform its job are listed in the parentheses that immediately follow the name are called arguments. The computer will retrieve the values held by the variables and hand them off to the function for its use. The number of arguments (or values) must match the variables in the definition by type and number. A semicolon must be present, when the function is called within the main() function.

The general form of a ANSI method of function call is as follows:

> *function_name* (*actual parameters*);

3.3.2.   Defining functions:

Defining a function means writing an actual code for the function which does a specific and identifiable task. Suppose you are defining a function which computes the square of a given number. Then you have to write a set of instructions to do this.

The general form of a ANSI method of function definition is as follows:

type_specifier function_name (*formal parameters*)
{
        variable declaration;   /* with in the function */
        body of function;
                *
                *
                *
        return (value_computed);
}

type_specifier specifies the type of value that the function's return statement returns. If nothing is returned to the calling function, then data type is **void**.

function_name is a user-defined function name. It must be a valid C identifier.

formal parameters is the type declaration of the variables of parameter list.

return is a keyword used to send the output of the function, back to the calling function. It is a means of communication from the called function to the calling
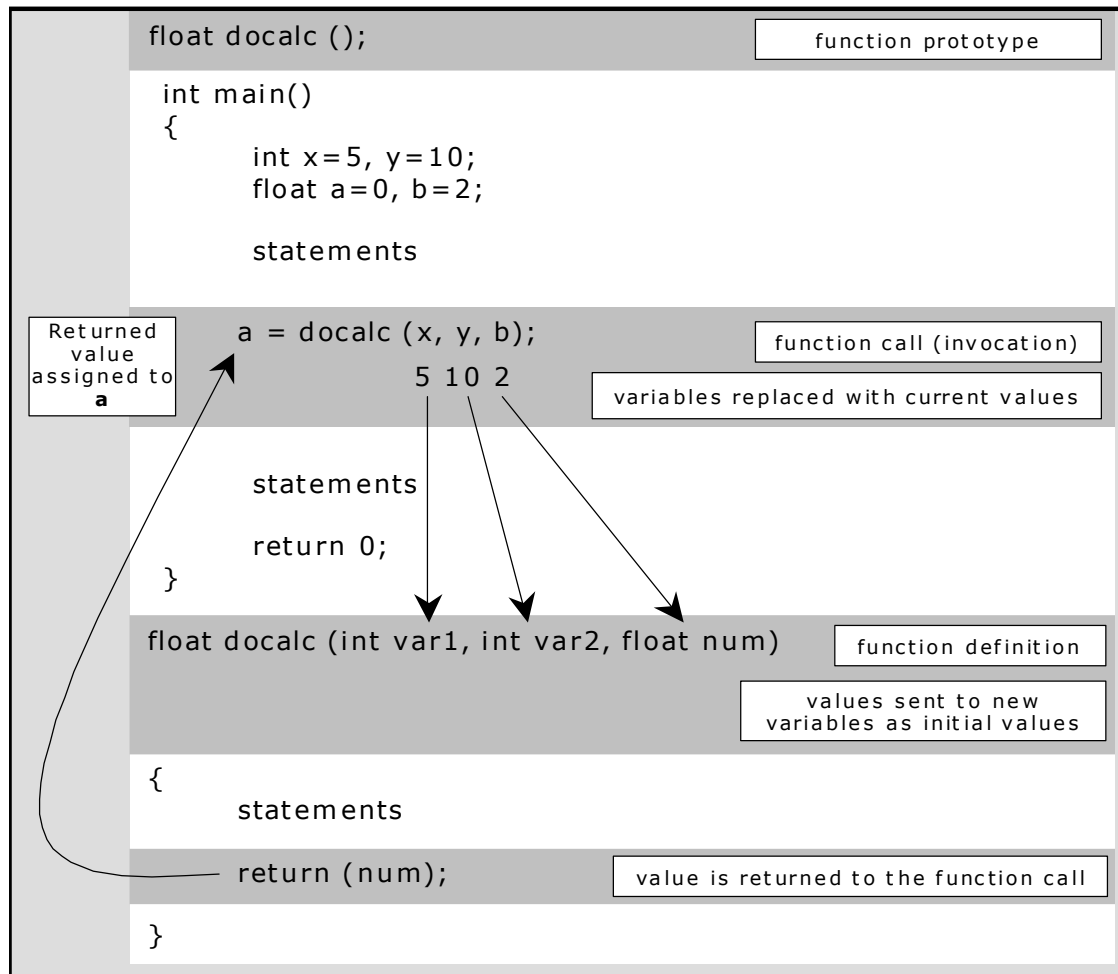
function. There may be one or more return statements. When a return is encountered, the control is transferred to the calling function.

{   is the beginning of the function.

}   is the end of function.

All the statements placed between the left brace and the corresponding right brace constitute the **body of a function**.

The following figure shows the flow through a function:

```
float docalc ();                                    function prototype

 int main()
 {
       int x=5, y=10;
       float a=0, b=2;

       statements

       a = docalc (x, y, b);                    function call (invocation)
                      5 10 2              variables replaced with current values


       statements

       return 0;
 }
float docalc (int var1, int var2, float num)      function definition
                                            values sent to new
                                            variables as initial values

 {
       statements

       return (num);                  value is returned to the function call

 }
```

Returned value assigned to **a**

Any values that the function receives are called parameters. It is easy to confuse arguments and parameters. The values are called arguments in the statement that calls the function but become parameters in the header of the function definition. A semicolon does not follow the function header.

Example:

This program accepts two integers and computes their sum via function **addnums**().

include <stdio.h>

int addnums(int, int);

```
main()
{
        int n1, n2, result;
        printf ("Enter two numbers:\n");
        scanf ("%d %d", &n1, &n2);
        result = addnums (n1, n2);                    /* Function call */
        printf ("The sum of %d and %d = %d\n", n1, n2, result);
}

int addnums (int val1, int val2)          /* Function definition to add two numbers */
{
        int sum;
        sum=val1 + val2;
        return (sum);
}
```

Output:

Enter two numbers: 12 34
The sum of 12 and 34 = 46

In the above example, n1 and n2 are two input values. These are passed to the function addnums(). In function definition, the copies of n1 and n2 are obtained, namely val1 and val2. After, the required computations are made in the function, the control returns to the calling function along with the value computed. In the above example, the main program receives the value of 46. This is then assigned to the variable result. Finally, the value of the result is displayed.

A function may return int, float, char or double type data to the main (). However, there is a restriction is returning arrays. Whenever an array is to be returned, the return statement is not required. If the return value is not an int then explicitly the data type of the return value should be specified.

3.3.3.   Function Declaration (Prototype):

Function declaration means specifying the function as a variable depending on the return value. It is declared in the declaration part of the main program. The default return value from a function is always an integer. If the function is returning a value other than an integer, it must be declared with the data type of the value it returns.

There is another reason for the function declaration. Functions may be written before or after the main(). If the functions are written before the main() then function declaration is not required. But, the functions which are written after the main() require function declaration. Function declaration informs the compiler, about the functions which are declared after the main().

A prototype may be created by copying the function header and pasting it above the main() definition and below the preprocessor directives (# include). A semicolon must follow a function prototype. The variable names do not need to be included in the prototype. If you include them, the compiler will ignore them.

The general form of function declaration in ANSI form is as follows:

Type_of_return_value function_name (parameter types);

Example:

1.   float average ();
2.   void message ();
3.   int max_nums ();

3.4. Function Arguments/Parameters:

Arguments and parameters are the variables used in a program and a function. Variables used in the calling function are called arguments. These are written within the parentheses followed by the name of the function. They are also called *actual parameters*, as they are accepted in the main program (or calling function).

Variables used in the function definition (called function) are called parameters. They are also referred to as *formal parameters*, because they are not the accepted values. They receive values from the calling function. Parameters must be written within the parentheses followed by the name of the function, in the function definition.

Parameter passing mechanisms:

A method of information interchange between the calling function and called function is known as parameter passing. There are two methods of parameter passing:

1.   Call by value

2.   Call by reference

3.4.1. Call by Value:

The call-by-value method copies the *value* of an argument into the formal parameter of the function. Therefore, changes made by the function to the parameters have no effect on the variables used to call it.

Example:

```
# include <stdio.h>

void swap (int x, int y);              /* function prototype */

main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (x, y);                   /* values passed */
}

void swap (int a, int b)               /* function definition */
{
        int temp;

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

3.4.2.   Call by Reference:

The call by reference method copies the *address* of an argument into the formal parameter. Inside the function, this address is used to access the argument used in the call. In this way, changes made to the parameter affect the variable used in the call to the function.

Call by reference is achieved by passing a pointer as the argument. Of course, in this case, the parameters must be declared as pointer types. The following program demonstrates this:

Example:

```
# include <stdio.h>

void swap (int *x, int *y);                    /* function prototype */

main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (&x, &y);                         /* addresses passed */
        printf ("%d %d\n", x, y);
}

void swap (int *a, int *b)
{
        int temp;

        temp = *a;
        *a = *b;
        *b = temp;
 }
```

The differences between call by value and call by reference:

Call by value	Call by reference
int a;	int a;
Formal parameter 'a' is a local variable.	Formal parameter is 'a' local reference.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

3.5.   Category of functions:

We have some other type of functions where the arguments and return value may be present or absent. Such functions can be categorized into:

1.   Functions with no arguments and no return value.

2.   Functions with arguments and no return value.

3.   Functions with no arguments but return value.

4.   Functions with arguments and return value.

3.5.1. Functions with no arguments and no return value:

Here, the called functions does not receive any data from the calling function and, it does not return any data back to the calling function. Hence, there is no data transfer between the calling function and the called function.

Example:

This program illustrates the function with no arguments and no return value.

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

main ()
{
        read_name ();
}
void read_value ()                          /*no return value */
{
        char name [10];
        printf ("Enter your name: ");
        scanf ("%s", name);
        printf ("\nYour name is %s: ", name);
}
```

Output:

```
        Enter your name: Herbert
        Your name is: Herbert
```

3.5.2. Functions with arguments and no return value:

Here, the called function receives the data from the calling function. The arguments and parameters should match in number, data type and order. But, the called function does not return and value back to the calling function. Instead, it prints the data in its scope only. It is one-way data communication between the calling function and the called function.

Example:

This program illustrates the function with arguments but has no return value.

```
#include <stdio.h>

void maximum (x, y);

main()
{
        int x, y;
        printf ("Enter the values of x and y: ");
        scanf ("%d %d", &x, &y);
        maximum (x, y);
}

void maximum (int p, int q)                 /* function to compute the maximum */
{
        if (p > q)
```

```
                printf ("\n maximum = %d", p);
        else
                printf ("\n maximum = %d" q);
        return;
}
```

Output:

Enter the values of x and y: 14 10
maximum = 14

3.5.3.   Function with no arguments but return value:

Here, the called function does not receive any data from the calling function. It manages with its local data to carry out the specified task. However, after the specified processing the called function returns the computed data to the calling function. It is also a one-way data communication between the calling function and the called function.

Example:

This program illustrates the function with no arguments but has a return value.

```c
#include <stdio.h>

float total ();

main ()
{
        float sum;
        sum = total ();
        printf ("Sum = %f\n", sum);
}

float total ()
{
        float x, y;
        x = 20.0;
        y = 10.0;
        return (x + y);
}
```

Output:

Sum = 30.000000

3.5.4.   Function with arguments and return value:

When a function has arguments it receives data from the calling function and does some process and then returns the result to the called function. In this way the main() function will have control over the function.

Example:

```
/* program to find the sum of first N natural numbers */

#include <stdio.h>
int sum (int x);                              /* function prototype*/

void main ()
{
        int n;
        printf ("Enter the limit: ");
        scanf ("%d", &n);
        printf ("sum of first %d natural numbers is: %d", n, sum(n));
}

int sum(int x)
{
        int i, result = 0
        for (i=1; i <= x; i++)
                result += i;
        return (result);
}
```

Output:

```
Enter the limit: 5
Sum of first 5 natural numbers is: 15
```

The main() is the calling function which calls the function sum(). The function sum() receives a single argument. Note that the called function (i.e., sum ()) receives its data from the calling function (i.e., main()). The return statement is employed in this function to return the sum of the n natural numbers entered from the standard input device and the result is displayed from the main() function to the standard output device. Note that int is used before the function name sum() instead of void since it returns the value of type int to the called function.

3.6.   Important points to be noted while calling a function:

- *Parenthesis are compulsory after the function name.*

- *The function name in the function call and the function definition must be same.*

- *The type, number, and sequence of actual and formal arguments must be same.*

- *A semicolon must be used at the end of the statement when a function is called.*

- *The number of arguments should be equal to the number of parameters.*

- *There must be one-to-one mapping between arguments and parameters. i.e. they should be in the same order and should have same data type.*

- *Same variables can be used as arguments and parameters.*

- *The data types of the parameters must match or be closely compatible. It is very risky to call a function using an actual parameter that is floating point data if the formal parameter was declared as an integer data type. You can pass a float to a double, but should not do the opposite. You can also pass a short int to an int, etc. But you should not pass any type of integer to any type of floating point data or do the opposite.*

3.7. Nested Functions:

We have seen programs using functions called only from the main() function. But there are situations, where functions other than main() can call any other function(s) used in the program. This process is referred as nested functions.

Example:

```c
#include <stdio.h>

void func1();
void func2();

void main()
{
        printf ("\n Inside main function");
        func1();
        printf ("\n Again inside main function");
}

void func1()
{
        printf ("\n Inside function 1");
        func2();
        printf ("\n Again inside function 1");
}

void func2()
{
        printf ("\n Inside function 2");
}
```

Output:

```
Inside main function
Inside function 1
Inside function 2
Again inside function 1
Again inside main function
```

Uses two functions func1() and func2() other than the main() function. The main() function calls the function func1() and the function func1() calls the function func2().

3.8.   Local and Global Variables:

Both the calling function and called function are using their own variables. The existence of these variables is restricted to the calling function or to the called functions. This is known as *scope of the variables*. Based on this scope the variables can be classified into:

1.   Local variables
2.   Global variables

Variables whose existence is known only to the main program or functions are called local variables. On the other hand, variables whose existence is known to both main() as well as other functions are called global variables. Local variables are declared within the main program or a function. But, the global variables are declared outside the main() and other functions.

The following program illustrates the concept of local and global variables:

Example:

```
#include <stdio.h>

int i = 10;                            /* global variable declaration */

main()
{
        int j;                         /* local variable declaration */
        printf ("i = %d\n", i);
        j = value (i);
        printf ("j = %d\n", j);
}
int value (int i)                      /* function to compute value */
{
        int k;                         /* local variable declaration */
        k = i +10
        return (k);
}
```

Output:

```
        i = 10
        j = 20
```

Distinguishing between local and global variables:

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

3.9.   Calling Functions with Arrays:

Generally, the C convention is the call-by-value. An exception is where arrays are passed as arguments. We can pass either an entire array as an argument or its individual elements.

Arrays in functions:

An array name represents the address of the first element in that array. So, arrays are passed to functions as pointers (a pointer is a variable, which holds the address of other variables).

When an array is passed to a function, only the *address* is passed, not the entire array. When a function is called with just the array name, a pointer to the array is passed. In C, an array name without an index is a pointer to the first element. This means that the parameter declaration must be of a compatible pointer type. There are three ways to declare a parameter which, is to receive an array pointer:

Example:

        void display (int num[10]);

in which case the compiler automatically converts it to a pointer.

As an unsized array:

        void display (int num[]);

which is also converted to a pointer.

As a pointer:

        void display (int *num);

which, is allowed because any pointer may be indexed with [] as if it were an array.

An array element used as an argument is treated like any other simple variable, as shown in the example below:

Example:

```
main ()
{
        . . .
      display (t [a]);
        . . .
}

void display (int num)
{
      printf ("%d", num);
 }
```

3.10. The return statement:

The return causes an immediate exit from a function to the point from where the function is called. It may also be used to return one value per call. All functions, except those of type void, return a value. If no return statement is present, most compilers will cause a 0 to be returned. The return statement can be any one of the types as shown below:

1.   return;

2.   return ();

3.   return (constant);

4.   return (variable);

5.   return (expression);

6.   return (conditional expression);

7.   return (function);

The first and second return statements, does not return any value and are just equal to the closing brace the function. If the function reaches the end without using a return statement, the control is simply transferred back to the calling portion of the program without returning any value. The presence of empty return statement is recommended in such situations.

The third return statement returns a constant function. For example:

        if (x <= 10)
                return (1);

The fourth return statement returns a variable to the calling function. For example:

        if (x <= 10)
                return (x);

The fifth return statement returns a value depending upon the expression specified inside the parenthesis. For example:

        return (a + b * c);

The sixth return statement returns a value depending upon the result of the conditional expression specified inside the parenthesis. For example:

        return (a > b ? 1 : 2);

The last return statement calls the function specified inside the parenthesis and collects the result obtained from that function and returns it to the calling function. For example:

        return (pow (4, 2));

The parenthesis used around the expression or value used in a return statement is optional.

3.11. User Defined Data Types:

C allows the programmer to create custom data types in five different ways. These are:

- The **structure**, which is a collection of variables under one name.

- Types created with **typedef**, which defines a new name for an existing type.

- The **union**, which enables the same piece of memory to be defined as two or more types of variables.

- The **enumeration**, which is a list of symbols.

- The **bit-field**, a variation of the structure, which allows easy access to the bits of a word.

3.11.1.   Structures:

A structure is a user defined data type. Using a structure we have the ability to define a new type of data considerably more complex than the types we have been using.

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. The variables which make up a structure are called **structure elements**.

In the following, the C keyword struct tells the compiler that a structure template is being defined:

```
struct addr
{
        char name[30];
        char street[40];
        int postalcode;
};
```

In this example, addr is known as the **structure tag**. A structure definition is actually a statement hence it is terminated by a semicolon.

A structure variable declaration of type addr is:

```
struct addr addr_info;
```

Variables may also be declared as the structure is defined. This can be done as follows:

```
struct addr
{
        char name[30];
        char street[40];
        int postalcode;
} addr_info, binfo, cinfo;
```

If only one variable is needed, the structure tag is not needed, for example:

```
struct
{
        char name[30];
        char street[40];
        int postalcode;
```

```
        } addr_info;
```

Structure Assignments:

ANSI C compilers allow the information in one structure to be assigned to another structure as:

```
        binfo = addr_info;
```

Declaring Structure Pointers:

Assuming the previously defined structure addr, the following declares addr_pointer as a pointer to data of that type:

```
        struct addr *addr_pointer;
```

Referencing Structure Elements:

Individual elements of a structure are referenced using the dot operator. The pointer to a structure member operator -> is used to access a member of a structure using a pointer variable.

Accessing Structure members using dot operator:

Individual elements of a structure are referenced using the dot operator, for example:

```
        addr_info. postalcode = 1234;
        printf ("%d",addr_info. postalcode);
```

Example:

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

struct
{
        char name[15];                  /*  childs name  */
        int  age;                       /*  childs age  */
        int  grade;                     /*  childs grade in school */
} boy, girl;

int main()
{
        strcpy (boy.name, "Herbert");
        boy.age = 15;
        boy.grade = 75;
        girl.age = boy.age - 1;         /* she is one year younger */
        girl.grade = 82;
        strcpy (girl.name, "Fousett");
        printf ("%s is %d years old and got a grade of %d\n",
                                                girl.name, girl.age, girl.grade);
        printf ("%s is %d years old and got a grade of %d\n",
                                                boy.name, boy.age, boy.grade);
        return 0;
}
```

Output:

Fousett is 14 years old and got a grade of 82
Herbert is 15 years old and got a grade of 75

Accessing Structure Pointers using -> operator:

Structure pointers may be used to generate a call by reference to a function and to create linked lists and other dynamic data structures. Call by reference can be used for structures to avoid overheads occurred by the push and pop operations of all the structure elements of the stack. For example if:

```
struct addr_info *ptr;
```

To access the elements of a structure using a pointer, the -> operator is used:

```
ptr -> postalcode
```

where, ptr has been declared as a pointer to the type of structure and assigned the address of a variable of that type, and postalcode is a structure element within that variable.

This can also be expressed as:

```
(*ptr). postalcode
```

The parenthesis is necessary because the structure member operator ". " takes higher precedence than does the indirection operator.

Example:

```c
#include <stdio.h>

struct
{
        char initial;
        int  age;
        int  grade;
} kids[12], *point, extra;

int main()
{
        int index;

        for (index = 0 ; index < 12 ; index++)
        {
                point = kids + index;
                point->initial = 'A' + index;
                point->age = 16;
                point->grade = 84;
        }

        kids[3].age = kids[5].age = 17;
        kids[2].grade = kids[6].grade = 92;
```

```
        kids[4].grade = 57;

        for (index = 0 ; index < 12 ; index++)
        {
                point = kids + index;
                printf("%c is %d years old and got a grade of %d\n",
                            (*point).initial, kids[index].age, point->grade);
        }
         extra = kids[2];                        /* Structure assignment */
         extra = *point;                         /* Structure assignment */

        return 0;
}
```

Output:

A is 16 years old and got a grade of 84
B is 16 years old and got a grade of 84
C is 16 years old and got a grade of 92
D is 17 years old and got a grade of 84
E is 16 years old and got a grade of 57
F is 17 years old and got a grade of 84
G is 16 years old and got a grade of 92
H is 16 years old and got a grade of 84
I is 16 years old and got a grade of 84
J is 16 years old and got a grade of 84
K is 16 years old and got a grade of 84
L is 16 years old and got a grade of 84

Arrays of Structures:

The following example declares a 100-element array of type addr:

        struct addr addr_arr [100];

in which the postalcode element of structure 3 would be accessed by:

        printf ("%d", addr_info [2]. postalcode);

Example:

```
#include <stdio.h>

struct
{
        char initial;
        int  age;
        int  grade;
} kids[12];

int main()
{
        int index;
        for (index = 0 ; index < 12 ; index++)
        {
```

```
            kids[index].initial = 'A' + index;
            kids[index].age = 16;
            kids[index].grade = 84;
        }

        kids[3].age = kids[5].age = 17;
        kids[2].grade = kids[6].grade = 92;
        kids[4].grade = 57;

        kids[10] = kids[4];                    /* Structure assignment  */

        for (index = 0 ; index < 12 ; index++)
            printf("%c is %d years old and got a grade of %d\n",
                        kids[index].initial, kids[index].age, kids[index].grade);
        return 0;
}
```

Output:

```
A is 16 years old and got a grade of 84
B is 16 years old and got a grade of 84
C is 16 years old and got a grade of 92
D is 17 years old and got a grade of 84
E is 16 years old and got a grade of 57
F is 17 years old and got a grade of 84
G is 16 years old and got a grade of 92
H is 16 years old and got a grade of 84
I is 16 years old and got a grade of 84
J is 16 years old and got a grade of 84
E is 16 years old and got a grade of 57
L is 16 years old and got a grade of 84
```

Passing Structures to Functions:

When an element of a non-global structure is to be passed to a function, the value of that element is passed (unless that element is complex, such as an array of characters).

Example:

```
        struct fred
        {
            char x;
            int y;
            char s[10];
        } mike;
```

each element would be passed like this:

```
        funct (mike.x);              /* passes character value of x  */
        funct (mike.y);              /* passes integer value of y  */
        funct (mike.s);              /* passes address of string  */
        funct (mike.s[2]);           /* passes character val of s[2]  */
```

If the address of the element is required to be passed, the & operator is placed before the structure name. In the above example, this would apply except in funct (mike.s);.

Passing entire Structures to Functions:

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that changes made to the contents of the structure inside the function do not affect the structure used as the argument.

Arrays and Structures with in Structures:

Structure elements may be arrays and structures. When a structure is an element of another structure, it is called a nested structure.
Example:

```c
#include <string.h>

struct person
{
        char name[25];
        int  age;
        char status;                    /* M = married, S = single */
};

struct alldat
{
        int grade;
        struct person descrip;
        char lunch[25];
};

int main()
{
        struct alldat student[53];
        struct alldat teacher, sub;
        teacher.grade = 94;
        teacher.descrip.age = 34;
        teacher.descrip.status = 'M';
        strcpy(teacher.descrip.name, "Mary Smith");
        strcpy(teacher.lunch, "Baloney sandwich");

        sub.descrip.age = 87;
        sub.descrip.status = 'M';
        strcpy(sub.descrip.name, "Old Lady Brown");
        sub.grade = 73;
        strcpy(sub.lunch, "Yogurt and toast");

        student[1].descrip.age = 15;
        student[1].descrip.status = 'S';
        strcpy(student[1].descrip.name, "Billy Boston");
        strcpy(student[1].lunch, "Peanut Butter");
        student[1].grade = 77;

        student[7].descrip.age = 14;
        student[12].grade = 87;

        return 0;
}
```

Output:

This program does not generate any output

Differences between arrays and structures:

Arrays	Structures
A array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.
Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we can dealing with structures.
Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are not stored in sequence of memory locations.
The array elements are accessed by its followed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.

Its general format is data type variable name [size];	Its general format is:
	struct <struct name>
	{
	    data_type structure member 1;
	    data_type structure member 2;
	         .
	         .
	         .
	    data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	    char studname [25];
	    int rollno;
	} stud1;

3.11.2.   typedef:

typedef allows a **new data type** to be explicitly defined. Note that it does not actually create a new data *class*, but simply defines a new name for an *existing* type. This process helps in making machine-dependent code more portable, since only the typedef statements need be changed. It can also aid in self-documenting a program by allowing the use of more descriptive names for the standard data types.

The general form of defining a new data type is:

        t*ypedef data type identifier*;

Where, identifier refers to new name(s) given to the data type. User defined types obey the same scope rules as identifiers, so if one defined in a function, it is recognized only within that function. Few examples of type definitions are

```
typedef int age;
typedef float average;
typedef char string;
```

Where *age* symbolizes int, *average* symbolizes float and *string* symbolizes char. They can be later used to declare variables as:

```
age child, adult;
average mark1, mark2;
string name[20];
```

Where *child* and *adult* are declared as integer variables, *mark1* and *mark2* are declared as floating point variables and name is declared as character array variable.

The typedef can also be used to define structures. The main advantage of type definition is that, you can create meaningful data type names for increasing the readability of the program. They also suggest the purpose of the data type names used in the program.

3.11.3.   Unions:

A **union** is a memory location, which is shared by two or more different variables, generally of different types, at different times. The general form of a union definition is:

```
union tag
{
        type variable_name;
        type variable_name;
        type variable_name;
                    .
                    .
} union_variables;
```

An example definition is as follows:

```
union u_type
{
        int i;
        char ch;
};
```

Declaration options are as for structures, i.e, variable names may be placed at the end of the definition or a separate declaration statement may be used.

A union shares the memory space instead of wasting storage on variables that are not being used. The compiler allocates a piece of storage that is large enough to hold the largest type in the union.

Accessing Union Elements:

union elements are accessed using the same methods used for accessing structure elements.

The union aids in the production of machine-independent code because the compiler keeps track of the actual sizes of the variables, which make up the union.

unions are frequently used when type conversions are needed because the data held in a union can be referred to in different ways.

Example 1:

```
union exam
{
        int roll_no;
        char name[15];
        int mark1,mark2,mark3;
} u1;

struct exam1
{
        int roll_no;
        char name[15];
        int mark1, mark2, mark3;
} s1;

void main()
{
        printf("The size of union is %d\n", sizeof(u1));
        printf("The size of structure is %d", sizeof(s1));
}
```

Output:

```
        The size of the union is 15
        The size of the structure is 23
```

In the above example, union *exam* has 5 members. The first member is a character array name having 15 characters (i.e., 15 bytes). The second member is of type int that requires 2 bytes for their storage. All the other members mark1, mark2, mark3 are of type int which requires 2 bytes for their storage. In the union, all these 5 members are allocated in a common place of memory (i.e., all the members share the same memory location). As union shares the memory space instead of wasting storage on variables that are not being used, the compiler allocates a piece of storage that is large enough to hold the largest type in the union. In the above declaration, the member name requires, 15 characters, which is the maximum of all members, hence a total memory space of 15 bytes is allocated. In case of structures, the total memory space allocated will be 23 (i.e. 15+2+2+2+2) bytes.

Example 2:

```
#include<stdio.h>
void main()
{
        union dec
```

```c
        {
                int x;
                char name[4];
        } dec1;
        int i;
        dec1.x = 300;
        printf("Size of union = %d\n", sizeof(dec1));
        printf("The value of x = %u\t", dec1.x);
        printf("\n%u %u %u %u", dec1.name[0], dec1.name[1], dec1.name[2],
                        dec1.name[3]);
}
```

Output:

Size of union = 4
The value of x = 300
44   1   65461   74

The binary value of 300 is 0000 0001 0010 1100. As per the internal storage representation first 0010 1100 = 44 is stored then 0000 0010 = 01 is stored.

Example 3:

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

struct hos
{
        char name[10];
        char hostelname[20];
};

struct daysch
{
        int phonenumber;
        char name[10];
        char address[40];
        int rollno;
};

union  student
{
        struct hos hostler;
        struct daysch dayscholar ;
} stu_data;

void main()
{
        int n;
        clrscr();
        printf("\n   MENU ");
        printf("\n 1. Hostler \n 2. Day Scholar\n");
        printf("\n enter the choice: ");
        scanf("%d",&n);
        if(n==1)
        {
```

```
                strcpy(stu_data.hostler.name,"Herbert");
                strcpy(stu_data.hostler.hostelname,"ss2");
                printf("\n student name: %s",stu_data.hostler.name);
                printf("\n hostle name: %s",stu_data.hostler.hostelname);
                printf("\n Union Data size: %d",sizeof(stu_data) );
                printf("\n Hostler Data size: %d",sizeof(stu_data.hostler) );
        }
        else if(n==2)
        {
                strcpy(stu_data.dayscholar.name,"Schildt");
                strcpy(stu_data.dayscholar.address,"balaji colony");
                stu_data.dayscholar.phonenumber=5620;
                stu_data.dayscholar.rollno = 1290;
                printf("\n student name: %s", stu_data.dayscholar.name);
                printf("\n address: %s", stu_data.dayscholar.address);
                printf("\n phone number: %d", stu_data.dayscholar.phonenumber);
                printf("\n roll number: %d", stu_data.dayscholar.rollno);
                printf("\n Union Data size: %d", sizeof(stu_data) );
                printf("\n Day Scholar Data size: %d", sizeof(stu_data.dayscholar) );
        }
        else
                printf("\n it is wrong choice ");
        getch();
}
```

Output:

RUN 1:

```
        MENU
  1.  Hostler
  2.  Day Scholar
```

Enter Your Choice: 2

Student name: Schildt
Address: balaji colony
Phone number: 5620
Roll no: 1290
Union data size: 54
Day scholar data size: 54

RUN 2:

```
        MENU
  1.  Hostler
  2.  Day Scholar
```

Enter Your Choice: 1

Student name: Herbert
Hostel name: ss2
Union data size: 54
Hostler data size: 30

In the above example, the declaration structure name hos requires 30 bytes and structure daysch requires 54 bytes of memory. In union stu_data both structures are allocated in a common place of memory (i.e. both are sharing same memory location). In our case a student can be either an hostler or a data scholar. So, the same memory

can be used for both type of students. So instead of allocating 84 bytes of memory the same can be achieved using 54 bytes of memory.

Differences between structures and unions:

Structures	Unions
Each member in a structure occupies and uses its own memory space	All the members of a union use the same memory space
The keyword struct tells us that we are dealing with structures	The keyword union tells us that we are dealing with unions
All the members of a structure can be initialized	Only the first member of an union can be initialized
More memory space is required since each member is stored in a separate memory locations	Less memory space is required since all members are stored int eh same memory locations

Its general format is:
struct <struct Name>
{
   data_type structure Member1;
   data_type structure Member2;
     .
     .
   data_type structure Member N;
} structure variable;
	Its general formula is:
	union <union Name>
	{
	   data_type structure Member1;
	   data_type structure Member2;
	     .
	     .
	   data_type structure Member N;
	} union variable;

Example:
struct student
{
   char studname[25];
   int rollno;
} stud1;
	Example:
	union student
	{
	   char studname[25];
	   int rollno;
	} stud1;

3.11.4. Enumerations:

The keyword **enum** is used to declare enumeration types. It provides a means of naming a finite set, and of declaring identifiers as elements of the set. Enumerations are defined much like structures. An example definition for user defined type day is as follows:

enum day {sun, mon, tue, wed, thu, fri, sat};

An example declaration is:

enum day day_of_week;

This creates the user defined type enum day. The keyword enum is followed by the tag name day. The enumerators are the identifiers sun, mon, . . ., sat. They are constants of type int. By default the first one 0 and each succeeding one has the next integer value.

Initialization:

The enumerators can be initialized. Also we can declare variables along with the template. The following example do so:

enum suit {clubs = 1, diamonds, hearts, spades} a, b, c;

clubs has been initialized to 1, diamonds, hearts and spades have the values 2, 3, and 4 respectively. a, b, and c are variables of this type.

As another example of initialization:

enum fruit {apple = 7, pear, orange = 3, lemon} frt;

Because the enumerator apple has been initialized to 7, pear has value 8. Similarly, because orange has a value 3, lemon has a value 4. Multiple values are allowed, but the identifiers themselves must be unique.

Example:

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

enum day {sun, mon, tue, wed, thu, fri, sat};

main()
{
        int n;
        clrscr ();
        printf ("Enter a day number (0 for sunday and so on upto 6):");
        scanf ("%d",&n);

        switch (n)
        {
                case sun:
                        printf("\nSunday");
                        break;
                case mon:
                        printf("\nMonday");
                        break;
                case tue:
                        printf("\nTuesday");
                        break;
                case wed:
                        printf("\nWednesday");
                        break;
                case thu:
                        printf("\nThursday");
                        break;
                case fri:
                        printf("\nFriday");
                        break;
                case sat:
                        printf("\nSaturday");
                        break;
        }
        getch();
}
```

3.11.5.   Bit-Fields:

C has a built-in method for accessing a single bit within a byte. This is useful because:

- If memory is tight, several Boolean (true/false) variables can be stored in one byte, saving memory.
- Certain devices transmit information encoded into bits within one byte.
- Certain encryption routines need to access individual bits within a byte.

Tasks involving the manipulation of bits may, of course, be performed using C's bitwise operators. However, the **bit-field** provides a means of adding more structure and efficiency to the coding.

Bit-Fields - A Type of Structure:

To access bits using bit-fields, C uses a method based on the structure. In fact, a bit-field is simply a special type of structure element, which defines how long in bits the field is to be.

The general form of a bit-field definition is:

```
        struct tag
        {
                type name1: length;
                type name2: length;
                type name3: length;
         } variable_list;
```

A bit-field must be declared as either int, unsigned, or signed. Bit-fields of length 1 should be declared as unsigned because a single bit cannot have a sign.

An Example Application:

Bit-fields are frequently used when analyzing input from a hardware device. For example, the status port of a serial communications device might return a status byte like this:

Bit	Meaning When Set
0	Change in clear-to-send line.
1	Change in data-set-ready.
2	Trailing edge detected.
3	Change in receive line.
4	Clear-to-send.
5	Data-set-ready.
6	Telephone ringing.
7	Received signal.

Defining the Bit-Field:

The foregoing can be represented in a status byte using the following bit-field definition/declaration:

```
struct status_type
{
        unsigned delta_cts: 1;
        unsigned delta_dsr: 1;
        unsigned tr_edge: 1;
        unsigned delta_rec: 1;
        unsigned cts: 1;
        unsigned dsr: 1;
        unsigned ring: 1;
        unsigned rec_line: 1;
} status;
```

Using this bit-field, the following routine would enable a program to determine whether it can send or receive data:

```
status = get_port_status;
if(status.cts)
        printf("Clear to send");
if(status.dsr)
        printf("Data set ready);
```

Referencing Bit-Field Elements:

Values are assigned to a bit-field using the usual method for assigning values to structure elements, e.g.:

```
status.ring = 0;
```

As with structures, if an element is referenced through a pointer, the -> operator is used in lieu of the dot operator.

Variations in Definition:

All bit-fields need not necessarily be named, which allows unused ones to be bypassed. For example, if, in the above example, access had only been required to cts and dsr, status_type could have been declared like this:

```
struct status_type
{
        unsigned: 4;
        unsigned cts: 1;
        unsigned dsr: 1;
} status;
```

Example:

```
#include <stdio.h>
union
{
      int index;
      struct
      {
              unsigned int x : 1;
              unsigned int y : 2;
```

```
                unsigned int z : 2;
        } bits;
} number;

int main ()
{
        for (number.index = 0 ; number.index < 20 ; number.index++)
        {
                printf ("index = %3d, bits = %3d%3d%3d\n", number.index,
                                        number.bits.z, number.bits.y, number.bits.x);
        }

        return 0;
}
```

Output:

```
index =   0, bits =   0  0  0
index =   1, bits =   0  0  1
index =   2, bits =   0  1  0
index =   3, bits =   0  1  1
index =   4, bits =   0  2  0
index =   5, bits =   0  2  1
index =   6, bits =   0  3  0
index =   7, bits =   0  3  1
index =   8, bits =   1  0  0
index =   9, bits =   1  0  1
index =  10, bits =   1  1  0
index =  11, bits =   1  1  1
index =  12, bits =   1  2  0
index =  13, bits =   1  2  1
index =  14, bits =   1  3  0
index =  15, bits =   1  3  1
index =  16, bits =   2  0  0
index =  17, bits =   2  0  1
index =  18, bits =   2  1  0
index =  19, bits =   2  1  1
```

Mixing Normal and Bit-Field Structure Elements:

Normal and bit-field structure elements may be mixed, as in:

```
        struct emp
        {
                struct addr address;
                float pay;
                unsigned lay-off: 1;          /*  lay-off or active  */
                unsigned hourly:  1;          /*  hourly pay or wage  */
                unsigned deducts: 3;          /*  tax deductions  */
        }
```

which demonstrates the use of only one byte to store information which would
otherwise require three bytes.

Limitations of bit-fields:

Bit-field variables have certain limitations. For example:

- The address of a bit-field variable cannot be taken.
- Bit-field variables cannot be arrayed.
- Integer boundaries cannot be overlapped.
- There is no certainty, from machine to machine, as to whether the fields will run from right to left or from left to right. In other words, any code using bit-fields may have some machine dependencies.

Key terms and Glossary

automatic: declared when entering the block, lost upon leaving the block; the declarations must be the first thing after the opening brace of the block

static: the variable is kept through the execution of the program, but it can only be accessed by that block

extern: available to all functions in the file AFTER the declaration; use extern to make the variable accessible in other files

register: automatic variable which is kept in fast memory; actual rules are machine-dependent, and compilers can often be more efficient in choosing which variables to use as registers.

Top-Down Design is an analysis method in which a major task is sub-divided into smaller, more manageable, tasks called functions. Each sub-task is then treated as a completely new analysis project. These sub-tasks may be sufficiently large and complex that they also require sub-division, and so on. The document that analysts use to represent their thinking related to this activity is referred to as a structure diagram.

Functions (also called as **procedures**, **modules**, or **subroutines**) are sections or blocks of code that perform steps or sub-steps in a program. Every C program contains at least one function. Its name must be "main" (all lowercase) and program execution always begins with it. Any function can call (execute) other functions. A function must be declared in the source code ahead of where it is called.

Predefined Functions are functions that are written and stored for use in future programs. Sometimes groups of these functions are organized by the type of purpose they perform and stored in header files as function libraries. For this reason, predefined functions are often referred to as library functions.

The **scope of an identifier** defines the area in program code where it has meaning. Identifiers that are defined *within* a function are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined *outside of* all functions are global, meaning that they will be recognized within *all* functions.

Call: The action of one function activating or making use of another functions is referred to as calling. A function must be declared in the source code ahead of the statement that calls it.

declaration: specifies the type of the identifier, so that it can subsequently be used.

definition: reserves storage for the object

The syntax for **declaring a function** is:

```
return-type function-name (argument declarations)
{
        local variable declarations
        statements
}
```

The function prototype is a declaration and is needed if the function is defined after its use in the program. The syntax is:

```
return-type function-name (argument declarations);
```

where, the argument declarations must include the types of the arguments, but the argument names are optional. If the function is defined before its use, then a prototype is not necessary, since the definition also serves as a declaration.

If the *return-type* is omitted, int is assumed. If there are no *argument declarations*, use void, not empty parentheses.

A **function prototype** is a statement (rather than an entire function declaration) that is placed ahead of a calling function in the source code to define the child function's label before it is used. The entire declaration must still be written later in the code, but may appear at a more convenient position following the calling function. A function prototype is simply a reproduction of a function header (the first line of a function declaration) that is terminated with a semi-colon.

Parameters are values that are passed between functions. The C statement that calls a function must include the function label followed by a parenthesized list of parameters that are being passed into the function. These parameters are referred to as **actual parameters** because they are the *actual* values being provided by the calling function. Actual parameters are often also referred to as **arguments**.

In the C code where the function is declared, a function header is written to declare the function identifier and to declare the identifiers, known as **formal parameters**, that will be used inside of the function to represent any data being passed into it. Formal parameters do not have to be identical to the actual parameters used in the calling statement, but must match in quantity and be compatible in data type.

The **scope of an identifier** defines the area in program code where it has meaning. Identifiers that are defined *within* a function are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined *outside of* all functions are global, meaning that they will be recognized within *all* functions.

References (labels) and **referencing** (use of those labels) involve the manner in which we use labels to refer to stored data.

Direct referencing is the use of a storage label to refer directly to the contents of a storage location. Direct referencing is often used in formulas when we want to use a stored value in the formula. The statement Y = X + 1; refers directly to the contents of the storage location X. The value stored in X will be added to 1.

Indirect referencing is the use of a storage label in such a way that the contents of one storage location (see "pointer" below) is used to refer *indirectly* to the contents of a different storage location. (See the web page about Pointers and Indirection.) Indirect referencing is often used to pass data back from a function to its parent when more than one parameter is involved. The statement *YP = 0; refers *indirectly* to the contents of a storage location that is pointed to by an address stored in YP. The identifier YP must be declared as a "pointer" (see below). The use of the star as an indirection operator in front of the identifier makes it an indirect reference. This operator also is often called the dereferencing operator because it prevents *direct reference* to the variable.

Address referencing is the use of a storage label to refer to its memory address rather than its contents. The statement scanf ("%d", &N); passes the *address* of storage location N to the scanf function. Had we used the identifier N without the preceding &, that would have passed the *contents* of the N storage location to the function instead of its address. A function can return a value of any type, using the return statement, The syntax is:

```
    return exp;
    return (exp);
    return;
```

The **return** statement can occur anywhere in the function, and will immediately end that function and return control to the function which called it. If there is no return statement, the function will continue execution until the closing of the function definition, and return with an undefined value.

A **structure** is a data type which puts a variety of pieces together into one object. The syntax is given below:

```
struct structure-tag-optional
{
        member-declarations
        structure-names-optional ;
}

struct structure-tag structure-name;

structure-name. member ptr-to-structure -> member
```

typedef: typedef defines a new type, which can simplify the code. Here is the syntax:

```
typedef data-type TYPE-NAME;

typedef struct structure-tag TYPE-NAME;

typedef struct
{
        member-declarations
} TYPE-NAME;
```

union: With union, different types of values can be stored in the same location at different times. Space is allocated to accomodate the largest member data type. They are syntactically identical to structures, The syntax is:

```
union union-tag-optional
{
        member-declarations
} union-names-optional;

union-name. member

ptr-to-union -> member
```

enum: The type enum lets one specify a limited set of integer values a variable can have. For example, flags are very common, and they can be either true or false. The syntax is:

```
enum enum-tag-optional {enum-tags} enum-variable-names-optional;

enum-name variable-name
```

The values of the enum variable are integers, but the program can be easier to read when using enum instead of integers. Other common enumerated types are weekdays and months.

Chapter 4

Pointers, Files, Command Line Arguments and Preprocessor

4.1. Pointers:

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

- It is the only way to express some computations.
- It produces compact and efficient code.
- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

C uses pointers explicitly with arrays, structures and functions.

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

The **unary** operator **&** gives the "address of a variable". The **indirection** or dereference operator ***** gives the "contents of an object **pointed to** by a pointer".

To declare a pointer to a integer variable do:

    int *pointer;

We must associate a pointer to a particular type. We can't assign the address of a short int to a long int.

Consider the effect of the following code:

```
#include <stdio.h>
main()
{
        int x = 1, y = 2;
        int *ip;
        ip = &x;
        y = *ip;
        *ip = 3;
}
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer works. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000 shown in figure 4.1.

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

        x   [    1    ]      y   [    2    ]      ip  [   100   ]
              100                  200                  1000
   y = *ip;

        x   [    1    ]      y   [    1    ]      ip  [   100   ]
              100                  200                  1000
   *ip = 3;

        x   [    3    ]      y   [    1    ]      ip  [   100   ]
              100                  200                  1000
```
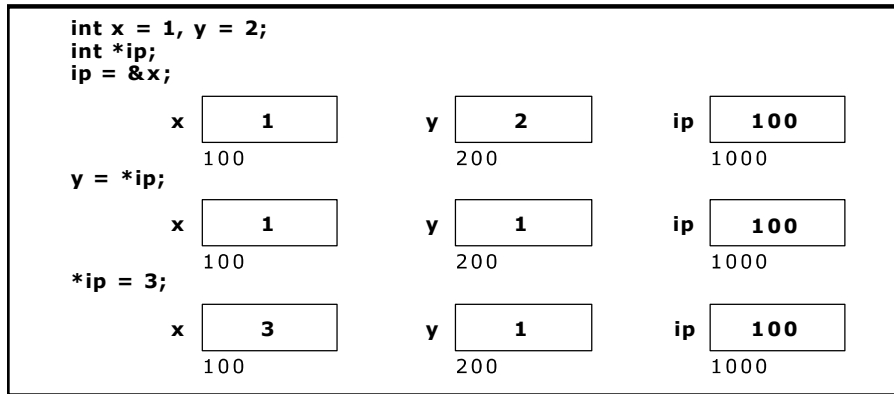
Fig. 4.1 Pointer, Variables and Memory

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

Next y gets assigned to the **contents of** ip. In this example ip currently **points** to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1. Finally, we can assign a value 3 to the contents of a pointer (*ip).

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

```
int *ip;
*ip = 100;
```

will generate an error (program crash!!). The correct usage is:

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

We can do integer arithmetic on a pointer, for example:

```
char m = 'j';
char *ch_ptr = &m;
float x = 20.55;
float *flp, *flq;
flp = &x;
*flp = *flp + 10;
++*flp;
(*flp)++;
flq = flp;
```

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

So, for a character pointer ++ch_ptr adds 1 byte to the address. For an integer or float ++ip or ++flp adds 2 bytes and 4 bytes respectively to the address.

Here is another **example** showing some of the uses of pointers:

```c
#include <stdio.h>

void main (void)
{
        int m = 0, n = 1, k = 2;
        int *p;
        char msg[] = "hello world";
        char *cp;
        p   = &m;                   /* p now points to m  */
        *p  = 1;                    /* m now equals 1 */
        k   = *p;                   /* k now equals 1 */
        cp  = msg;                  /* cp points to the first character of msg */
        *cp = 'H';                  /* change the case of the 'h' in msg      */
        cp  = &msg[6];              /* cp points to the 'w' */
        *cp = 'W';                  /* change its case  */
        printf ("m = %d, n = %d, k = %d\nmsg = \"%s\"\n", m, n, k, msg);
}
```

Output:

m = 1, n = 1, k = 1
msg = "Hello World"

Note the very important point that the name of an array (`msg' in the above example), if used without an index, is considered to be a pointer to the first element of the array.

In fact, an array name followed by an index is exactly equivalent to a pointer followed by an offset.

Example:

```c
#include <stdio.h>

void main (void)
{
        char msg[] = "hello world";
        char *cp;
        cp = msg;
        cp[0]      = 'H';
        *(msg+6)  = 'W';
        printf ("%s\n", msg);
        printf ("%s\n", &msg[0]);
        printf ("%s\n", cp);
        printf ("%s\n", &cp[0]);
}
```

Output:

Hello World
Hello World
Hello World
Hello World

Note, however, that `cp' is a variable, and can be changed, whereas `msg' is a constant, and is not an lvalue.

4.1.1. Pointers and Arrays:

There is a close association between pointers and arrays. Let us consider the following statements:

    int x[5] = {11, 22, 33, 44, 55};
    int *p = x;

The array initialization statement is familiar to us. The second statement, array name x is the starting address of the array. Let we take a sample memory map as shown in figure 4.2.:

From the figure 4.2 we can see that the starting address of the array is 1000. When x is an array, it also represents an address and so there is no need to use the (&) symbol before x. We can write int *p = x in place of writing int *p = &x[0].

The content of p is 1000 (see the memory map given below). To access the value in x[0] by using pointers, the indirection operator * with its pointer variable p by the notation *p can be used.

	Memory	
1000	11	x[0]
1002	22	x[1]
1004	33	x[2]
1006	44	x[3]
1008	55	x[4]
1010		
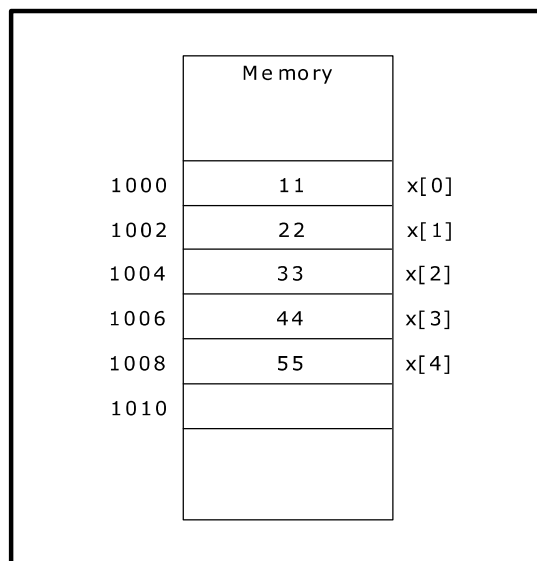
Figure 4.2. Memory map - Arrays

The increment operator ++ helps you to increment the value of the pointer variable by the size of the data type it points to. Therefore, the expression p++ will increment p by 2 bytes (as p points to an integer) and new value in p will be 1000 + 2 = 1002, now *p will get you 22 which is x[1].

Consider the following expressions:

    *p++;
    *(p++);
    (*p)++;

How would they be evaluated when the integers 10 & 20 are stored at addresses 1000 and 1002 respectively with p set to 1000.

p++   : The increment ++ operator has a higher priority than the indirection operator
         * . Therefore p is increment first. The new value in p is then 1002 and the
         content at this address is 20.

*(p++): is same as *p++.

(*p)++: *p which is content at address 1000 (i.e. 10) is incremented. Therefore (*p)++ is 11.

Note that, *p++ = content at incremented address.

Example:

#include <stdio.h>

```
main()
{
        int x[5] = {11, 22, 33, 44, 55};
        int *p = x, i;                          /*  p=&x[0] = address of the first element
*/
        for (i = 0; i < 5; i++)
        {
                printf ("\n x[%d] = %d", i, *p);              /* increment the address*/
                p++;
        }
}
```
Output:

x [0] = 11
x [1] = 22
x [2] = 33
x [3] = 44
x [4] = 55

The meanings of the expressions p, p+1, p+2, p+3, p+4 and the expressions *p, *(p+1), *(p+2), *(p+3), *(p+4) are as follows:

P = 1000 P+1 = 1000 + 1 x 2 = 1002 P+2 = 1000 + 2 x 2 = 1004 P+3 = 1000 + 3 x 2 = 1006 P+4 = 1000 + 4 x 2 = 1008	*p = content at address 1000 = x[0] *(p+1) = content at address 1002 = x[1] *(p+2) = content at address 1004 = x[2] *(p+3) = content at address 1006 = x[3] *(p+4) = content at address 1008 = x[4]

4.1.2.   Pointers and strings:

A string is an array of characters. Thus pointer notation can be applied to the characters in strings. Consider the statements:

        char tv[20] = "ONIDA";
        char *p = tv;

For the first statement, the compiler allocates 20 bytes of memory and stores in the first six bytes the char values as shown below:

Variable	tv[0]	tv[1]	tv[2]	tv[3]	tv[4]	tv[5]
Value	'O'	'N'	'I'	'D'	'A'	'\0'
Address	1000	1001	1002	1003	1004	1005

The statement:

        char *p = tv;   /* or p = &tv[0] */

Assigns the address 1000 to p. Now, we will write a program to find the length of the string tv and print the string in reverse order using pointer notation.

Example:

```
#include <stdio.h>

main()
{
        int n, i;
        char tv[20] = "ONIDA";              /* p = 1000 */
        char *p = tv, *q;                   /* p = &tv[0], q is a pointer */
        q = p;
        while (*p != '\0')          /* content at address of p is not null character */
                p++;
        n = p - q;                          /* length of the string */
        --p;                   /* make p point to the last character A in the string */
        printf ("\nLength of the string is %d", n);
        printf ("\nString in reverse order: \n");
        for (i=0; i<n; i++)
        {
                putchar (*p);
                p--;
        }
}
```

Output:

Length of the string is 5
String in reverse order: ADINO

4.1.3.   Pointers and Structures:

You have learnt so far that pointers can be set to point to an int, float, char, arrays and strings. Now, we will learn how to set pointers to structures. Consider the structure definition.

        struct student
        {
                int rollno;
                char name [20];
        };

and the declarations:

        struct student s;
        struct student *ps = &s;

in the last statement, we have declared a pointer variable ps and initialized it with the address of the structure s. We say that ps points to s. To access the structure members with the help of the pointer variable ps, we use the following syntax:

ps → rollno   (or)   (*ps).rollno
ps → name   (or)   (*ps).name

The symbol → is called *arrow operator* and is made up of a minus sign and a greater than sign. The parentheses around ps are necessary because the member operator (.) has a higher precedence than the indirection operator (*).

We will now write a program to illustrate the use of structure pointers.

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

struct invent
{
        char name[20];
        int number;
        float price;
};

main()
{
        float temp;
        struct invent product[3], *ps;
        int size ;
        ps = &product[0];
        printf("input product details:");
        size = sizeof(product[0]);
        printf("\n sizeof(product[0]) = %d",size );
        printf("\n product = %u ",product);
        printf("\n &product[0] = %u ",&product[0]);
        printf("\n &product[1] = %u ",&product[1]);
        printf("\n &product[2] = %u ",&product[2]);
        printf("\nproduct + 3 = %u\n",(product+3) );
        printf("\n Name \t Number \t Price\n");
        for (ps=product; ps < product+3; ps++)
        {
                scanf ("%s %d %f", ps->name, &ps->number, &temp);
                ps->price = temp;
        }
        printf("\n Item Details...\n Name\tNumber\t Price\n");
        for (ps=product; ps < product+3; ps++)
                printf ("\n%s %d %f", ps->name, ps->number, ps->price);
        getch();
}
```

Output:

input Product details:

sizeof(product[0]) = 26
product = 9478
&product[0] = 9478
&product[1] = 9504
&product[2] = 9530

product + 3 = 9556

Name   Number Price
Pen   101   10.45
Pencil 102   5.67
Book   103   12.63

Item Details…….

Name   Number Price
Pen   101   10.45
Pencil 102   5.67
Book   103   12.63

The compiler reserves the memory locations as shown below:

	← product[0] →			← product[1] →			← product[2] →	
name	number	price	name	number	price	name	number	price

Address 9478   9498 9500 9504   9524 9526 9530   9550 9552
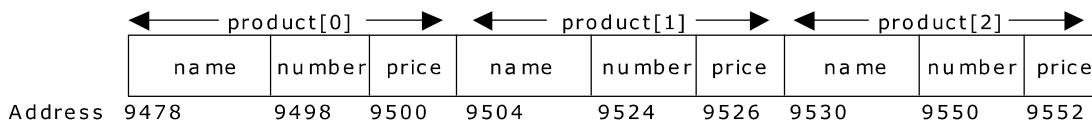
4.1.4.   Pointer and Function:

Pointer has deep relationship with function. We can pass pointer to the function and also addresses can be passed to the function as pointer argument. This can be represented by two ways as defined below:

1.   Pointer as function argument.

2.   Pointer to function.

4.1.4.1.   Pointer as function argument:

This is achieved by call by reference. The call by reference method copies the *address* of an argument into the formal parameter. Inside the function, this address is used to access the argument used in the call. In this way, changes made to the parameter affect the variable used in the call to the function.

Call by reference is achieved by passing a pointer as the argument. Of course, in this case, the parameters must be declared as pointer types. The following program demonstrates this:

Example:

```
# include <stdio.h>

void swap (int *x, int *y);           /* function prototype */

main ()
{
      int x, y;
      x = 10;
      y = 20;
      swap (&x, &y);                  /* addresses passed */
      printf ("%d %d\n", x, y);
}
```

```
void swap (int *a, int *b)
{
        int temp;

        temp = *a;
        *a = *b;
        *b = temp;
}
```

4.1.4.2.   Pointer to Function:

A function works like a variable. It is also similar to data type declaration like the other variable. Pointer to a function variable occurs when you define a function as a pointer. As function has address location in memory the syntax used to declare pointer to function is as:

> *Return _type (*function name) (list of arguments);*
>
> > or
>
> *return_type (*fptr) (void);*
>
> > or
>
> *void (*fptr) (float, int);*

Where, return_type is the data type which may be integer, float or character and *(fptr) (argument list) is pointer to function. The procedure to illustrate this concept is as follows:

```
void add (int, int);
void (*f1) (int, int);
f1 = & add;
```

By using the above example, we say that add() is a simple function. But f1 is pointer variable which work as pointer to function and third statement the address of the add() function is assigned to f1 variable which is a memory variable. So, (*f1)(int, int) is a function pointer linked to the function add.

Example:

/* Program to add and substract two numbers by using the pointer to function. */

```
#include <stdio.h>

void add (int, int);
void sub (int, int);
void (*f1) (int, int);

main()
{
        f1 = & add;
        (*f1) (10, 15);
        f1 = & sub;
        (*f1) (11, 7);
}

void add (int a, int b)
```

```
{
        printf ("\n Sum = %d", a + b);
}

void sub (int a, int b)
{
        printf("\n sub = %d", a - b);
}
```

Output:

Sum = 25
Sub = 4

4.1.5.   Array of Pointers:

We can have array of pointers since pointers are variables. *Array of Pointers* will handle variable length text lines efficiently and conveniently. How can we do this is:

Example:

```
#include <stdio.h>

void main(void)
{
        char *p[5];
        int x;
        p[0] = "DATA";
        p[1] = "LOGIC";
        p[2] = "MICROPROCESSOR";
        p[3] = "COMPUTER";
        p[4] = "DISKETTE";
        for (x = 0; x < 5; ++x)
                printf("%s\n", p[x]);
}
```

Output:

DATA
LOGIC
MICROPROCESSOR
COMPUTER
DISKETTE

4.1.6.   Multidimensional arrays and pointers:

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

        a[n][m] notation.

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows are irrelevant. The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

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

We can do:

```
fun (int a[][35])
{
        ... statements...
}
```

or even:

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

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

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

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

Now lets look at the difference between pointers and arrays. Strings are a common application of this. Consider:

char *name[10];

char aname[10][20];

- **aname** is a <u>true</u> 200 elements 2D char array.

- **name** has 10 pointer elements.

The advantage of the latter is that each pointer can point to arrays of different length. Let us consider (shown in figure 4.3):

char *name[10] = {"no month", "jan", "feb"};

char aname[10][20] = {"no month", "jan", "feb"};
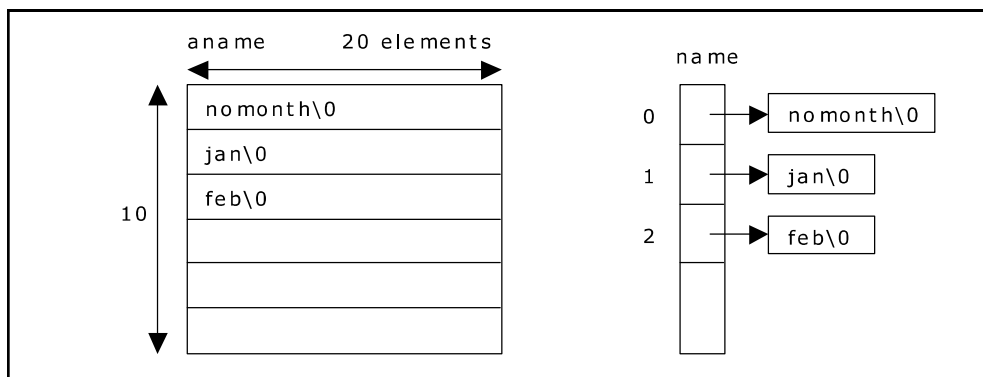
Fig. 4.3. 2D Arrays and Arrays of Pointers

4.1.7.　　Pointer to Pointer:

The concept of a pointer having the exclusive storage address of another pointer is called as pointer to pointer. Normally we do not use pointers to pointers very often but they can exist. The following program demonstrates this:

Example:

```
#include <stdio.h>

void main(void)
{
        int x, *p, **ptp;
         x = 454;
         p = &x;
         ptp = &p;
         printf("%d  %d\n", *p, **ptp);
}
```

Output:

454   454

**ptp is declared as a pointer to pointer of type int. This means that ptp handles the address of another pointer p. Variable x is assigned a value of 454, then its address is assigned to pointer p. Next, the address of pointer p is assigned to pointer to a pointer ptp. **ptp accesses the value of x.

4.1.8.　　Illegal indirection:

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

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

Let us have a pointer: char *p;

For example, let us consider:

```
1.      *p = (char *) malloc(100);          /* request 100 bytes of memory */
2.      *p = `y';
```

There is a mistake above. That is, * before p in the statement 1. The correct code should be:
```
p = (char *) malloc(100);
```

If no memory is available for allocation p will be NULL. Therefore we can't do:

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

A good C program should check for this:

```
p = (char *) malloc (100);
if ( p == NULL)
{
```

```
        printf("Error: Out of Memory\n");
        exit(1);
}
*p = 'y';
```

4.1.9.   Dynamic Memory Allocation and Dynamic Structures:

Dynamic allocation is a unique feature to C among high level languages. It enables us to create data types and structures of any size and length to suit our program need. The process of allocating memory at run time is known as dynamic memory allocation.

The four important memory management functions for dynamic memory allocation and reallocation are:

1. malloc
2. calloc
3. free
4. realloc

The malloc function:

The function malloc is used to allocate a block of memory of specified size. It is defined by:
        void *malloc (number_of_bytes)

The malloc function returns a pointer to the first byte of the allocated memory block.

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

        char *cp;
        cp = (char *) malloc (100);

attempts to get 100 bytes and assigns the starting address to cp.

We can also use the sizeof() function to specify the number of bytes. For example

        int *ip;
        ip = (int *) malloc (100*sizeof(int));

Some C compilers may require to cast the type of conversion. The (int *) means force to an integer pointer. Forcing to the correct pointer type is very important to ensure pointer arithmetic is performed correctly.

It is good practice to use sizeof() even if you know the actual size you want - it makes for device independent (portable) code.

The sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function. So:

        int i;
        struct complex
        {
                int real;
                int imaginary;
        };

typedef struct complex comp;

sizeof(int), sizeof(i),
sizeof(struct complex) and
sizeof(comp) are all ACCEPTABLE.

The free function:

It frees (releases) the memory space allocated for a block. The syntax is:

free (ptr);

This releases the block of memory allocated which is currently pointed to by the pointer variable ptr. The advantage is simply memory management when we no longer need a block.

The calloc and realloc functions:

There are two additional memory allocation functions, calloc() and realloc(). Their prototypes are given below:

void *calloc (num_elements, element_size};

void *realloc (void *ptr, new_size);

malloc does not initialise memory (to **zero**) in any way. If you wish to initialise memory then use calloc. The calloc is slightly more computationally expensive but, occasionally, more convenient than malloc. The syntax difference between calloc and malloc is that calloc takes the number of desired elements, num_elements, and element_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

int *ip;
ip = (int *) calloc (100, sizeof(int));

The realloc is a function, which attempts to change the size of a previous allocated block of memory by malloc function. The new size can be larger or smaller. If the block is made larger than the older, it will extend only if it can find additional space in the same region. Otherwise, it may create entirely a new region (again, if space is available) and move the contents of the old block into the new block without any loss.

The realloc function to increase the size of block to 200 integers instead of 100, simply do:
ip = (int *) calloc (ip, 200);
If the size is to be made smaller then the remaining contents are unchanged. The realloc function to reduce the size of block to 50 integers instead of 200, simply do:

ip = (int *) calloc (ip, 50);

Example:

include <stdio.h>
include <alloc.h>
include <string.h>

```
main()
{
        char *ptr, *qtr;
        ptr = (char *) malloc ( 12);
        strcpy(ptr, "hello world" );
        printf("\n Block now contains: %s", ptr);
        qtr =(char *) realloc(ptr,25);
        strcpy(qtr, "Hello beautiful world");
        printf("\n The Block contents after reallocation: %s", qtr);
        ptr =(char *) realloc(qtr, 5);
        printf("\n After reducing the size: %s", ptr);
}
```

Output:

Block now contains: hello world
The Block contents after reallocation: Hello beautiful world
After reducing the size: Hello

The differences between the address stored in a pointer and a value at the address:

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address, which itself is stored in a pointer.

The differences between the indirection operator and the address of operator:

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

4.2.   Files:

File is a bunch of bytes stored in a particular area on some storage devices like floppy disk, hard disk, magnetic tape and cd-rom etc., which helps for the permanent storage.

There are 2 ways of accessing the files. These are:

- Sequential access: the data can be stored or read back sequentially.
- Random access: the data can be access randomly.

If a file can support random access (sometimes referred to as position requests), opening a file also initializes the file position indicator to the start of a file. This indicator is incremented as each character is read from, or written to, the file.

The close operation disassociates a file from a specific stream. If the file was opened for output, the close operation will write the contents (if any) of the stream to the device. This process is usually called flushing the stream.

All files are closed automatically when the program terminates, but not when it crashes.
Each stream associated with a file has a file control structure of type *FILE*.

4.2.1. Streams:

Even though different devices are involved (terminals, disk drives, etc), the buffered file system transforms each into a logical device called a **stream**. Because streams are device-independent, the same function can write to a disk file or to another device, such as a console. There are two types of streams:

> **Text Streams:** A text stream is a sequence of characters. In a text stream, certain character translations may occur (for example, a newline may be converted to a carriage return/line-feed pair). This means that there may not be a one-to-one relationship between the characters written and those in the external device.

> **Binary Streams:** A binary stream is a sequence of bytes with a one-to-one correspondence to those on the external device (i.e, no translations occur). The number of bytes written or read is the same as the number on the external device. However, an implementation-defined number of bytes may be appended to a binary stream (for example, to pad the information so that it fills a sector on a disk).

4.2.2. File Input and Output functions:

The ANSI file system comprises several interrelated functions. These are:

Function	Description
fopen()	Opens a file.
fclose()	Closes a file.
putc()	Writes a character.
fputc()	Writes a character.
getc()	Reads a character.
fgetc()	Reads a character.
fseek()	Seeks a specified byte in a file.
fprintf()	Is to a file what printf() is to the console.
fscanf()	Is to a file what scanf() is to a console.
feof()	Returns TRUE if end-of-file is reached.
ferror()	Returns TRUE if an error is detected.
rewind()	Resets file position to beginning of file.
remove()	Erases a file.
fflush()	Flushes a file.

Most of these functions begin with the letter "f". The header file stdio.h provides the prototypes for the I/O function and defines these three types:

```
typedef  unsigned long    size_t
typedef  unsigned long    fpos_t
typedef  struct _FILE      FILE
```

stdio.h also defines the following:

```
EOF             -1          /* value returned at end of file */
SEEK_SET         0          /* from beginning of file  */

SEEK_CUR         1          /* from current position  */
SEEK_END         2          /* from end of file  */
```

The latter three are used with fseek() function which performs random access on a file.

4.2.3.   The File Pointer:

C treats a file just like a stream of characters and allows input and output as a stream of characters. To store data in file we have to create a buffer area. This buffer area allows information to be read or written on to a data file. The buffer area is automatically created as soon as the file pointer is declared. The general form of declaring a file is:

        FILE *fp;

FILE is a defined data type, all files should be declared as type FILE before they are used. FILE should be compulsorily written in upper case. The pointer fp is referred to as the stream pointer. This pointer contains all the information about the file, which is subsequently used as a communication link between the system and the program.

A file pointer fp is a variable of type FILE that is defined in stdio.h.

4.2.4.   Opening a File:

fopen() opens a stream for use, links a file with that stream and returns a pointer associated with that file. The prototype of fopen() function is as follows:

        FILE *fopen (const char * *filename*, const char * *mode*);

Where, filename is a pointer to a string of characters that make a valid filename (and may include a path specification) and mode determines how the file will be opened. The legal values for mode are as follows:

Value	Description
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append a binary file for read/write.

A file may be opened in text or binary mode. In most implementations, in text mode, CR/LF sequences are translated to newline characters on input. On output, the reverse occurs. No such translation occurs on binary files.

The following opens a file named TEST for writing:

```
FILE *fp;
fp = fopen ("test", "w");
```

However, because fopen() returns a null pointer if an error occurs when a file is opened, this is better written as:

```
FILE *fp;
if ((fp = fopen ("test", "w")) == NULL)
{
        printf("cannot open file\n");
        exit(1);
}
```

4.2.5.   Closing a File:

fclose() closes the stream, writes any data remaining in the disk buffer to the file, does a formal operating system level close on the file, and frees the associated file control block. fclose() has this prototype:

```
int fclose (FILE *fp);
```

A return value of zero signifies a successful operation. Generally, fclose() will fail only when a disk has been prematurely removed or a disk is full.

4.2.6.   Writing a Character:

Characters are written using putc() or its equivalent fputc(). The prototype for putc() is:

```
int putc (int ch, FILE *fp);
```

where ch is the character to be output. For historical reasons, ch is defined as an int, but only the low order byte is used.

If the putc() operation is successful, it returns the character written, otherwise it returns EOF.

4.2.7.   Reading a Character:

Characters are read using getc() or its equivalent fgetc(). The prototype for getc() is:

```
int getc(FILE *fp);
```

For historical reasons, getc() returns an integer, but the high order byte is zero. getc() returns an EOF when the end of file has been reached. The following code reads a text file to the end:

```
        do
        {
                ch = getc (fp);
        } while(ch != EOF);
```

4.2.8.   Using feof():

As previously stated, the buffered file system can also operate on binary data. When a file is opened for binary input, an integer value equal to the EOF mark may be read, causing the EOF condition. To solve this problem, C includes the function feof(), which determines when the end of the file is reached when reading binary data.

The prototype is:

        int feof (FILE *fp);

The following code reads a binary file until end of file is encountered:

        while (! feof (fp))
                ch = getc(fp);

Of course, this method can be applied to text files as well as binary files.

4.2.9.   Working With Strings - fputs() and fgets():

In addition to getc() and putc(), C supports the related functions fputs() and fgets(), which read and write character strings. They have the following prototypes:

        int fputs (const char *str, FILE *fp);

        char *fgets (char *str, int length, FILE *fp);

The function fputs() works like puts() but writes the string to the specified stream. The fgets() function reads a string until either a newline character is read or length-1 characters have been read. If a newline is read, it will be part of the string (unlike gets()). The resultant string will be null-terminated.

4.2.10.   rewind ():

rewind() resets the file position indicator to the beginning of the file. The syntax of rewind() is:

        rewind(fptr);

where, fptr is a file pointer.
4.2.11.   ferror ():

ferror() determines whether a file operation has produced an error. It returns TRUE if an error has occurred, otherwise it returns FALSE. ferror() should be called immediately after each file operation, otherwise an error may be lost.

4.2.12.   Erasing Files:

remove () erases a specified file. It returns zero if successful.

4.2.13.   Flushing a Stream:

fflush () flushes the contents of an output stream. fflush() writes the contents of any unbuffered data to the file associated with fp. It returns zero if successful.

4.2.14.   fread () and fwrite ():

To read and write data types which are longer than one byte, the ANSI standard provides fread() and fwrite(). These functions allow the reading and writing of blocks of any type of data. The prototypes are:

> size_t fread (void *buffer, size_t num_bytes, size_t count, FILE *fp);

> size_t fwrite (const void *buffer, size_t num_bytes, size_t count, FILE *fp);

For fread(), buffer is a pointer to a region of memory which will receive the data from the file. For fwrite(), buffer is a pointer to the information which will be written. The buffer may be simply the memory used to hold the variable, for example, &l for a long integer.

The number of bytes to be read/written is specified by num_bytes. count determines how many items (each num_bytes in length) are read or written.

fread() returns the number of items read. This value may be less than count if the end of file is reached or an error occurs. fwrite() returns the number of items written.

One of the most useful applications of fread() and fwrite() involves reading and writing user-defined data types, especially structures. For example, given this structure:

```
struct struct_type
{
        float balance;
        char name[80];
} cust;
```

The following statement writes the contents of cust:

> fwrite (&cust, sizeof(struct struct_type), 1, fp);

4.2.15.   fseek() and Random Access I/O:

Random read and write operations may be performed with the help of fseek(), which sets the file position locator. The prototype is:

> int fseek(FILE *fp, long numbytes, int origin);

in which numbytes is the number of bytes from the origin, which will become the new current position, and origin is one of the following macros defined in stdio.h:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End-of-file	SEEK_END

fseek() returns 0 when successful and a non-zero value if an error occurs. fseek() may be used to seek in multiples of any type of data by simply multiplying the size of the data by the number of the item to be reached, for example:

        fseek (fp, 9*sizeof (struct list), SEEK_SET);

Which seeks the tenth address.

4.2.16.   fprint() and fscanf():

fprint() and fscanf() behave exactly like print() and scanf() except that they operate with files. The prototypes are:

        int fprintf (FILE *fp, const char *control_string, ...);
        int fscanf (FILE *fp, const char *control_string, ...);

Although these functions are often the easiest way to read and write assorted data, they are not always the most efficient. Because formatted ASCII data is being written as it would appear on the screen (instead of in binary), extra overhead is incurred with each call. If speed or file size is of concern, use fread() and fwrite().

4.2.17.   The Standard Streams:

Whenever a C program starts execution, three streams are opened automatically.
These are:

• Standard input (stdin)
• Standard output (stdout)
• Standard error (stderr)

Normally, these streams refer to the console, but they may be redirected by the operating system to some other device or environment. Because the standard streams are file pointers, they may be used to provide buffered I/O operations on the console, for example:

```
putchar(char c)
{
      putc(c, stdout);
}
```


4.3.   Command Line Arguments:

Some times it is very useful to pass information into a program when we run it from the command prompt. The general method to pass information into main() function is through the use of command line arguments.

A command line argument is the information that follows the program's name on the command prompt of the operating system.

        For example: TC program_name

There are three special built_in_arguments to main(). They are:

• The first argument is argc (**arg**ument **c**ount) must be an integer value, which represents the number arguments in the command prompt. It will

always be at least one because the name of the program qualifies as the first argument.

- The second argument argv (**arg**ument **v**ector) is a pointer to an array of strings.

- The third argument env (**env**ironment data) is used to access the DOS environmental parameters active at the time the program begins execution.

When an array is used as an argument to function, only the address of the array is passed, not a copy of the entire array. When you call a function with an array name, a pointer to the first element in the array is passed into a function. (In C, an array name without as index is a pointer to the first element in the array).
Each of the command line arguments must be separated by a space or a tab. If you need to pass a command line argument that contains space, you must place it between quotes as:

"this is one argument"

Declaration of argv must be done properly, A common method is:

char *argv[];

That is, as a array of undetermined length.

The env parameter is declared the same as the argv parameter, it is a pointer to an array of strings that contain environmental setting.

Example 4.3.1:

```
/*      The following program will print "hello tom".      */

# include <stdio.h>
# include <process.h>

main( int argc, char *argv[])
{
        if(argc!=2)
        {
                printf("You forgot to type your name\n");
                exit(0);
        }
        printf("Hello %s ", argv[1]);
}
```

if the name of this program is name.exe, at the command prompt you have to type as:

➢ name tom

Output:

hello tom

Example 4.3.2:

```
/*      This program prints current environment settings        */
# include <stdio.h>

main (int argc, char *argv[], char *env[])
{
        int i;
        for(i=0; env[i]; i++)
                printf("%s\n", env[i]);
}
```

We must declare both argc and argv parameters even if they are not used because the parameter declarations are position dependent.

4.4.   Example Programs on File I/O and Command Line Arguments:

The following programs demonstrate the use of C's file I/O functions.

Example 4.4.1:

Program on fopen(), fclose(), getc(), putc(). Specify filename on command line. Input chars on keyboard until $ is entered. File is then closed. File is then re-opened and read.

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

void main(int argc, char *argv[])
{
        FILE *fp;                               /* file pointer  */
        char ch;

        if(argc!=2)
        {
                printf("You forgot to enter the file name\n");
                exit(1);
        }

        if((fp=fopen(argv[1], "w"))==NULL)              /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
        do                                      /* get keyboard chars until `$'  */
        {
                ch = getchar();
                putc(ch, fp);
        } while (ch! = '$');

        fclose(fp);                             /* close file  */

        if((fp=fopen(argv[1], "r"))==NULL)              /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
```

```
        ch = getc(fp);                          /* read one char  */

        while(ch != EOF)
        {
                putchar(ch);                    /* print on screen  */
                ch = getc(fp);                  /* read another char */
        }

        fclose(fp);                             /* close file  */
}
```

Example 4.4.2:

Program on feof() to check for EOF condition in Binary Files. Specify filenames for input and output at command prompt. The program copies the source file to the destination file. feof() checks for end of file condition. The feof() can also be used for text files.

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

void main(int argc, char *argv[])
{
        FILE *in, *out;                         /* file pointers  */
        char ch;

        if(argc != 3)
        {
                printf("You forgot to enter the filenames\n");
                exit(1);
        }

        if((in=fopen(argv[1], "rb"))==NULL)     /* open source file  */
        {                                       /* for read binary  */
                printf("Cannot open source file\n");
                exit(1);
        }

        if((out=fopen(argv[2], "wb"))==NULL)     /* open dest file  */
        {                                        /* for write binary  */
                printf("Cannot open destination file\n");
                exit(1);
        }
        while(! feof(in))                        /* here it is  */
        {
                ch = getc(in);

                if(! feof(in))                   /* and again  */
                        putc(ch, out);
        }

        fclose(in);                              /* close files  */
        fclose(out);
}
```

Example 4.4.3:

Program on fputs(), fgets and rewind(). Strings are entered from keyboard until blank line is entered. Strings are then written to a file called 'testfile'. Since gets() does not store the newline character, '\n' is added before the string is written so that the file can be read more easily. The file is then rewind, input and displayed.

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

void main(void)
{
        FILE *fp;                               /* file pointer  */
        char str[80];

        if((fp=fopen("testfile", "w+"))==NULL)     /* open file for text read & write  */
        {
                printf("Cannot open file\n");
                exit(1);
        }

        do                              /* get strings until CR and write to file  */
        {
                printf("Enter a string (CR to quit):\n");
                gets(str);
                strcat(str, "\n");
                fputs(str, fp);
        } while(*str != '\n');

        rewind(fp);                             /* rewind  */

        while(! feof(fp))                               /* read and display file  */
        {
                fgets(str, 79, fp);
                printf(str);
        }

        fclose(fp);                             /* close file  */
}
```

Example 4.4.4:

Program on fread() and fwrite() (for Data Types Longer than Byte) which writes, then reads back, a double, an int and a long. Notice how sizeof () is used to determine the length of each data type. These functions are useful for reading and writing user-defined data types, especially structures.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{

        FILE *fp;
        double d=12.23;
        int i=101;
        long l=123023;
```

```
        if((fp=fopen("testfile", "wb+"))==NULL)        /* open for binary read & write
*/
        {
                printf("Cannot open file\n");
                exit(1);
        }

/* parameters are: *buffer, number of bytes, count of items, file pointer  */

        fwrite(&d, sizeof(double), 1, fp);
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&l, sizeof(long), 1, fp);

        rewind(fp);

        fread(&d, sizeof(double), 1, fp);
        fread(&i, sizeof(int), 1, fp);
        fread(&l, sizeof(long), 1, fp);

        printf("%2.3f   %d   %ld\n",d,i,l);

        fclose(fp);
}
```

Example 4.4.5:

Program on fprintf() and fscanf(). Reads a string and an integer from the keyboard, writes them to a file, then reads the file and displays the data. These functions are easy to write mixed data to files but are very slow compared with fread() and fwrite().

```
#include <stdio.h>
#include <stdlib.h>
#include <exec/io.h>

void main (void)
{

        FILE *fp;
        char s[ 80];
        int t;
        if ((fp=fopen("testfile", "w"))==NULL)                /* open for text write  */
        {
                        printf ("Cannot open file\n");
                        exit (1);
        }

        printf ("Enter a string and a number: ");

                        /* parameters are: FILE *fp, const char *control_string, ...  */

        fscanf (stdin, "%s%d", s, &t);
        fprintf (fp, "%s %d", s, t);
        fclose (fp);

        if ((fp=fopen("testfile", "r"))==NULL)                /* open for text read */
        {
                printf ("Cannot open file\n");
```

```
            exit (1);
        }

        fscanf (fp,"%s%d",s,&t);
        fprintf (stdout, "%s %d\n", s, t);

        fclose (fp)
}
```

4.5.   The C Preprocessor:

As defined by the ANSI standard, the C preprocessor contains the following directives:

#if	#ifdef	#ifndef	#else	#elif	#include
#define	#undef	#line	#error	#pragma	#define

#define:

Defines an identifier (the macro name) and a string (the macro substitution), which will be substituted for the identifier each time the identifier is encountered in the source file.

Once a macro name has been defined, it may be used as part of the definition of other macro names.

If the string is longer than one line, it may be continued by placing a backslash on the end of the first line.

By convention, C programmers use uppercase for defined identifiers. Example macro #defines are:

```
#define TRUE 1
#define FALSE 0
```

The macro name may have arguments, in which case every time the macro name is encountered, the arguments associated with it are replaced by the actual arguments found in the program, as in:

```
#define ABS(a) (a)<0 ? -(a) : (a)
        ...
printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));
```

Such macro substitutions in place of real functions increase the speed of the code at the price of increased program size.

#error

#error forces the compiler to stop compilation. It is used primarily for debugging. The general form is:

```
#error  error_message
```

When the directive is encountered, the error message is displayed, possibly along with other information (depending on the compiler).

#include

#include instructs the compiler to read another source file, which must be included between double quotes or angle brackets. Examples are:

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

Both instruct the compiler to read and compile the named header file.

If a file name is enclosed in angle brackets, the file is searched for the included file in a special known \include directory or directories. If the name is enclosed in double quotes, the file is searched in the current directory. If the file is not found, the search is repeated as if the name had been enclosed in angle brackets.

4.5.1.   Conditional Compilation:

Several directives control the selective compilation of portions of the program code. They are:

```
#if            #else          #elif          #endif
```

The general form of **#if** is:

```
#if constant_expression
        statement sequence
#endif
```

#else works much like the C keyword else. #elif means "else if" and establishes an if-else-if compilation chain.

Amongst other things, #if provides an alternative method of "commenting out" code. For example, in

```
#if 0
        printf("#d", total);
#endif
```

the compiler will ignore printf("#d", total);.

#ifdef and #ifndef

#ifdef means "if defined", and is terminated by an #endif. #indef means "if not defined".

#undef

#undef removes a previously defined definition.

#line

#line changes the contents of __LINE__ (which contains the line number of the currently compiled code) and __FILE__ (which is a string which contains the name of the source file being compiled), both of which are predefined identifiers in the compiler.

#pragma

The #pragma directive is an implementation-defined directive which allows various instructions to be given to the compiler.

Example 4.5.1.1:

```
/*      program  to demonstrate #ifdef,  #ifndef , #else  and #endif    */

# include <stdio.h>
# include <conio.h>
# define  a 10

main()
{
        #ifdef a
                printf("\n Hello a is defined..");
        #endif

        #ifndef a
                printf("\n Not defined ");
        #else
                printf("\n defined ");
        #endif

        getch();
}
```

Output:

Hello a is defined..
defined

4.5.2.   The # and ## Preprocessor Operators:

The # and ## preprocessor operators are used when using a macro #define.

The # operator turns the argument it precedes into a quoted string. For example, given:

```
        #define mkstr(s) # s
```

the preprocessor turns the line:

```
        printf (mkstr (I like C);
```
into:
```
        printf ("I like C");
```

The ## operator concatenates two tokens. For example, given:

```
        #define concat(a, b) a ## b
        int xy=10;
        printf("%d", concat (x, y));
```

the preprocessor turns the last line into:
```
        printf("%d", xy);
```

4.5.3.   Mathematics: <math.h>

Mathematics is relatively straightforward   library to use again. You **must** include #include <math.h>. Below we list some common math functions.

double acos(double x);	Compute arc cosine of x.
double asin(double x);	Compute arc sine of x.
double atan(double x);	Compute arc tangent of x.
double atan2(double y, double x);	Compute arc tangent of y/x.
double ceil(double x);	Get smallest integral value that exceeds x.
double cos(double x);	Compute cosine of angle in radians.
double cosh(double x);	Compute the hyperbolic cosine of x.
div_t div(int number, int denom);	Divide one integer by another.
double exp(double x);	Compute exponential of x
double fabs (double x);	Compute absolute value of x.
double floor(double x);	Get largest integral value less than x.
double fmod(double x, double y);	Divide x by y with integral quotient and return remainder.
double frexp(double x, int *expptr);	Breaks down x into mantissa and exponent of no.
labs(long n);	Find absolute value of long integer n.
double ldexp(double x, int exp);	Reconstructs x out of mantissa and exponent of two.
ldiv_t ldiv(long number, long denom);	Divide one long integer by another.
double log(double x);	Compute log(x).
double log10 (double x);	Compute log to the base 10 of x.
double modf(double x, double *intptr);	Breaks x into fractional and integer parts.
double pow (double x, double y);	Compute x raised to the power y.
double sin(double x);	Compute sine of angle in radians.
double sinh(double x);	Compute the hyperbolic sine of x.
double sqrt(double x);	Compute the square root of x.
void srand(unsigned seed);	Set a new seed for the random number generator (rand).
double tan(double x);	Compute tangent of angle in radians.
double tanh(double x);	Compute the hyperbolic tangent of x.

Math Constants:

The math.h library defines many (often neglected) constants. It is always advisable to use these definitions:

HUGE	The maximum value of a single-precision floating-point number.
M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.

M_LN2	The natural logarithm of 2.
M_LN10	**The natural logarithm of 10.**
M_PI	п.
M_PI_2	п/2.
M_PI_4	п/4.
M_1_PI	1/ п.
M_2_PI	2/ п.
M_2_SQRTPI	$2 / \sqrt{\pi}$.
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.
MAXFLOAT	The maximum value of a non-infinite single-precision floating point number.
HUGE_VAL	positive infinity.

There are also a number a machine dependent values defined in #include <value.h>.

4.5.4.   Character conversions and testing: ctype.h

The library #include <ctype.h> which contains many useful functions to convert and test **single** characters. The common functions are prototypes are as follows:

Character testing:

int isalnum(int c);	True if c is alphanumeric.
int isalpha(int c);	True if c is a letter.
int isascii(int c);	True if c is ASCII.
int iscntrl(int c);	True if c is a control character.
int isdigit(int c);	True if c is a decimal digit
int isgraph(int c);	True if c is a graphical character.
int islower(int c);	True if c is a lowercase letter
int isprint(int c);	True if c is a printable character
int ispunct (int c);	True if c is a punctuation character.
int isspace(int c);	True if c is a space character.
int isupper(int c);	True if c is an uppercase letter.
int isxdigit(int c);	True if c is a hexadecimal digit

Character Conversion:

int toascii(int c);	Convert c to ASCII.
Int tolower(int c);	Convert c to lowercase.
int toupper(int c);	Convert c to uppercase.

The use of these functions is straightforward.

4.5.5.   Memory Operations:

The library #include <memory.h> contains the basic memory operations. Although not strictly string functions the functions are prototyped in #include <string.h>:

void *memchr (void *s, int c, size_t n);	Search for a character in a buffer.
int memcmp (void *s1, void *s2, size_t n);	Compare two buffers.
void *memcpy (void *dest, void *src, size_t n);	Copy one buffer into another.
void *memmove (void *dest, void *src, size_t n);	Move a number of bytes from one buffer to another.
void *memset (void *s, int c, size_t n);	Set all bytes of a buffer to a given character.

Note that in all case to **bytes** of memory are copied. The sizeof() function comes in handy again here, for example:

```
char src[SIZE], dest[SIZE];
int  isrc[SIZE], idest[SIZE];
```

memcpy(dest, src, SIZE);	Copy chars (bytes)
memcpy(idest, isrc, SIZE*sizeof(int));	Copy arrays of ints
memmove() behaves in exactly the same way as memcpy() except that the source and destination locations may overlap.	
memcmp() is similar to strcmp() except here **unsigned bytes** are compared and returns less than zero if s1 is less than s2 **etc.**	

4.6.   String Searching:

The library also provides several string searching functions:

char *strchr (const char *string, int c);	Find first occurrence of character c in string.
char *strrchr (const char *string, int c);	Find last occurrence of character c in string.
char *strstr (const char *s1, const char *s2);	locates the first occurrence of the string s2 in string s1.
char *strpbrk (const char *s1, const char *s2);	returns a pointer to the first occurrence in string s1 of any character from string s2, or a null pointer if no character from s2 exists in s1
size_t strspn (const char *s1, const char *s2);	returns the number of characters at the begining of s1 that match s2.
size_t strcspn (const char *s1, const char *s2);	returns the number of characters at the begining of s1 that **do not** match s2
char *strtok (char *s1, const char *s2);	break the string pointed to by s1 into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by s2.
char *strtok_r (char *s1, const char *s2, char **lasts);	has the same functionality as strtok () except that a pointer to a string placeholder lasts must be supplied by the caller.
strchr () and strrchr () are the simplest to use.	

Example 4.6.1:

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

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strchr (str1,'l');
        printf("%s\n", ans);
}
```

Output: llo

After this execution, ans points to the location str1 + 2

strpbrk () is a more general function that searches for the first occurrence of any of a group of characters, for example:

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

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strpbrk (str1,"aeiou");
        printf("%s\n",ans);
}
```

Output: ello

Here, ans points to the location str1 + 1, the location of the first e.

strstr () returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1.

Example 4.6.2:

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

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strstr (str1, "lo");
        printf("%s\n", ans);
 }
```

Output: lo   /*   will yield ans = str + 3.   */

strtok () is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to strtok() will start from this position

if on these subsequent calls the first argument is NULL. For example, If we wish to break up the string str1 at each space and print each token on a new line we could do:

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

main()
{
        char *str1 = "Hello Big Boy";
        char *t1;
        for(t1 = strtok (str1," "); t1 != NULL; t1 = strtok (NULL," "))
                printf("%s\n", t1);
 }
```

Output:

Hello
Big
Boy

The initialisation calls strtok () loads the function with the string str1.We terminate when t1 is NULL. We keep assigning tokens of str1 to t1 until termination by calling strtok () with a NULL first argument.

SAMPLE PROGRAMS ON POINTERS

Example 1:

```
#include <stdio.h>
void main(void)
{
        int i, *p;
        i = 43;
        p = & i;
        printf ("%d  %d\n", i, *p);
}
```

Output:

43 43

Example 2:

```
#include <stdio.h>
void main(void)
{
        int i, *p;
        i = 43;
        p = &i;
        *p = 16;
        printf("%d\n", i);
}
```

Output:

16

Example 3:

```
#include <stdio.h>

void change(int *i);

void main(void)
{
        int x;
        x = 146;
        change(&x);
        printf("%d\n", x);
}

void change(int *x)
{
        *x = 351;
}
```

Output:

351

Example 4:

```
#include <stdio.h>
void swap(int *i, int *ii);
void main(void)
{
        int x, y;
        x = 10;
        y = 127;

        swap(&x, &y);
        printf("%d  %d\n", x, y);
}

void swap(int *a, int *b)
{
        int i;
        i = *a;
        *a = *b;
        *b = i;
}
```

Output:

```
127  10
```

Example 5:

```
#include <stdio.h>
void main(void)
{
        void *v;
        char *c;
        c = "Testing void pointer";
        v = c;
        printf("%s\n", v);
}
```

Output:

```
Testing void pointer
```

Example 6:

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

void main(void)
{
        char a[40], *p;
        strcpy(a, "DATA");
        p = a;
        printf("%s\n", p);
}
```

Output:

```
DATA
```

Example 7:

```
#include <stdio.h>
void main(void)
{
        char *a;
        int x;
        a = "DATA";
        for (x = 0; x <= 4; ++x)
                printf("%c\n", *(a + x));
}
```

Output:

```
D
A
T
A
```

Example 8:

```
#include <stdio.h>
void main(void)
{
        char *a;
        a = "DATA";
        while (*a != '\0')
        printf("%c\n", *a++);
}
```

Output:

```
D
A
T
A
```

Example 9:

```
#include <stdio.h>

void main(void)
{
        char *a;
        a = "DATA";
        while (*a)
                printf("%c\n", *a++);
}
```

Output:

```
D
A
T
A
```

Example 10:

```
#include <stdio.h>
void main(void)
{
        char *a;
        a = "DATA";
        printf("%u\n", a);
        while (*a)
                printf("%c\n", *a++);
        printf("%u\n", a);
}
```

Output:

```
118
D
A
T
A
122
```

Example 11:

```
#include <stdio.h>
void main(void)
{
        char *p[5];
        int x;
        p[0] = "DATA";
        p[1] = "LOGIC";
        p[2] = "MICROPROCESSOR";
        p[3] = "COMPUTER";
        p[4] = "DISKETTE";
        for (x = 0; x < 5; ++x)
                printf("%s\n", p[x]);
}
```

Output:

```
DATA
LOGIC
MICROPROCESSOR
COMPUTER
DISKETTE
```

Example 12:

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

void main(void)
{
 char *a, b[256];
 printf("enter a string:");
```

```
gets(b);
a = (char*) malloc(strlen(b));
if (a == NULL)
{
        printf("Out of memory\n");
        exit(0);
}
else
        strcpy(a, b);
puts(a);
getch();
}
```

Output:

```
Enter a string: aabbc
aabbc
```

Example 13:

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

void main(void)
{
        char *a, b[256];
        printf("enter a string:");
       gets(b);
        if ((a = (char*) malloc(strlen(b))) == NULL)
        {
                printf("Out of memory\n");
                exit(0);
        }
        else
                strcpy(a, b);
puts(a);
 getch();
}
```

Output:

```
Enter a string: aabbc
aabbc
```

Example 14:

```
#include <stdio.h>
void main(void)
{
        int x[40];
        *(x + 0) = 65;
        printf("%c\n", *(x + 0));
}
```

Output:

```
A
```

Example 15:

```
#include <stdio.h>

int square(int i, *ii);

void main(void)
{
        int x, y;
        x = 8;
        y = 11;
        x = square(x, &y);
        printf("%d  %d\n", x, y);
}

int square(int a, int *b)
{
        *b = *b * *b;                /* rewrite memory */
        return(a * a);               /* return square */
}
```

Output:

    64 121

Example 16:

```
#include <stdio.h>
void square(int *i, int *j);

void main(void)
{
        int x, y;
      x = 8;
        y = 11;
        square(&x, &y);
        printf("%d  %d\n", x, y);
}

void square(int *a, int *b)
{
        *a *= *a;
        *b *= *b;
}
```

Output:

    64 121

Example 17:

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

char *combine(char *, char *);
```

```c
void main(void)
{
        char a[10], b[10], *p;
        strcpy(a, "house");
        strcpy(b, "fly");
        p = combine(a, b);
        printf("%s\n", p);
}

char *combine(char *s, char *t)
{
        int x, y;
        char r[100];
        strcpy(r, s);
        y = strlen(r);
        for (x = y; *t != '\0'; ++x)
                r[x] = *t++;
        r[x] = '\0';
        return(r);
}
```

Output:

housefly

Example 18:

```c
struct test
{
        int a;
        char b;
        char name[20];
};

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

void load(struct test *);

void main(void)
{
        struct test r;
        load(&r);
        printf("%d  %c  %s\n", r.a, r.b, r.name);
}

void load(struct test *s)
{
        s->a = 14;
        s->b = 'A';
        strcpy(s->name, "Group");
}
```

Output:

14 A Group

Example 19:

```
struct e_record
{
        char name[15];
        int id;
        int years;
        double sal;
};

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

void main(void)
{
        struct e_record e, *emp;
        emp = &e;
        strcpy(emp->name, "Herbert Schildt");
        emp->id = 14;
        emp->years = 22;
        emp->sal = 5305.00;
        printf("Employee Name: %s\n", emp->name);
        printf("Employee Number: %d\n", emp->id);
        printf("Years Employed: %d\n", emp->years);
        printf("Employee Salary: Rs.%-.2lf\n", emp->sal);
}
```

Output:

```
Employee Name: Herbert Schildt
Employee Number: 14
Years Employed: 22
Employee Salary: Rs. 5305.00
```


Example 20:

```
struct e_record
{
        char name[15];
        int id;
        int years;
        double sal;
};

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

void main(void)
{
        struct e_record *emp;
        if ((emp = (e_record*) malloc(sizeof(struct e_record))) == NULL)
                exit(0);
        strcpy(emp->name, "Herbert Schildt");
        emp->id = 14;
        emp->years = 22;
        emp->sal = 535.00;
```

```
            printf("Employee Name: %s\n", emp->name);
            printf("Employee Number: %d\n", emp->id);
            printf("Years Employed: %d\n", emp->years);
            printf("Employee Salary: Rs.%-.2lf\n", emp->sal);
}
```

Output:

```
Employee Name: Herbert Schildt
Employee Number: 14
Years Employed: 22
Employee Salary: Rs. 5305.00
```

Example 21:

```
#include <stdlib.h>
void main(void)
{
        int *x;
        float *y;
        double *z;
    if ((z = malloc(sizeof(double))) == NULL)
        exit(0);
        y = (float *) z;
        x = (int *) z;
                        /* x, y, and z all contain the same address */
                        /* memory allocation is for a double type */
        *x = 19;
        *y = 131.334;
        *z = 9.9879;
}
```

Output: This program does not generate any output.

Example 22:

```
#include <stdlib.h>
union test
{
        int x;
        float y;
        double z;
};
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        union test *ptr;
        if ((ptr = malloc(sizeof(union test))) == NULL)
                exit(0);
        printf("%d\n", sizeof(union test));
        ptr->x = 19;
        ptr->y = 131.334;
        ptr->z = 9.9879;
}
```

Output: 8

Example 23:

```c
struct e_record
{
        char name[15];
        int id;
        int years;
        double sal;
};

struct a_record
{
        char name[15];
        char address[40];
};

union com
{
        struct e_record a;
        struct a_record b;
};

#include <stdio.h>
#include <string.h>
void main(void)
{
        union com v;
        strcpy(v.a.name, "Bill Collins");
        v.a.id = 44;
        v.a.years = 12;
        v.a.sal = 351.22;
        printf("Name: %s\n", v.a.name);
        printf("Identification: %d\n", v.a.id);
        printf("Tenure: %d years\n", v.a.years);
        printf("Salary: $%-.2lf\n\n", v.a.sal);
        strcpy(v.b.name, "Bill Collins");
        strcpy(v.b.address, "523 Short St.; Front Royal, VA  22630");
        printf("Name: %s\n", v.b.name);
        printf("Address: %s\n", v.b.address);
}
```

Example 24:

```c
#include <stdio.h>
void main(void)
{
        int x, *p, **ptp;
        x = 454;
        p = &x;
        ptp = &p;
        printf("%d  %d\n", *p, **ptp);
}
```

Output:

454   454

Example 25:

```c
#include <stdio.h>
#include <string.h>
void main(void)
{
        char a[20], *b, **c;
        strcpy(a, "Capacitor");
        b = a;
        c = &b;
        printf("%s  %s\n", b, *c);
}
```

Output:

Capacitor   Capacitor

Example 26:

```c
#include <stdio.h>

void main(void)
{
        int x, *p, **ptp, ***ptptp, ****ptptptp;
        x = 274;
        p = &x;
        ptp = &p;
        ptptp = &ptp;
        ptptptp = &ptptp;
        printf("%d\n", ****ptptptp);
}
```

Output:

274

IMPORTANT REVIEW QUESTIONS

1. Distinguish between getchar() and gets() functions.

getchar()	gets()
Used to receive a single character.	Used to receive a single string with white spaces.
Does not require any argument.	It requires a single argument.

2. Distinguish between scanf() and gets() functions.

scanf()	gets()
Strings with spaces cannot be accessed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character array data can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers, float etc. can be read.	Only one string can be read at a time.

3. Distinguish between printf() and puts() functions.

puts()	printf()
It can display only one string at a time.	It can display any number of characters, integers or strings at a time.
No such conversion specifications. Every thing is treated as string.	Each data type is considered separately depending upon the conversion specifications.

4. What is the difference between a pre increment and a post increment operation?

A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1.

5. Distinguish between break and continue statement.

Break	Continue
Used to terminate the loops or to exit from loop or switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop.	The continue statement when executed cause immediate termination of the current iteration of the loop.

6. Distinguish between while and do-while loops.

While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

7. Distinguish between local and global variables

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

8. State the differences between the function prototype the function definition

Function prototype	Function Definition
It declares the function.	It defines the function.
It ends with a semicolon.	It doesn't ends with a semicolon.
The declaration need not include parameters.	It should include names for the parameters.

9. Compare and contrast recursion and iteration

Both involve repetition.
Both involve a termination test.
Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation condition fails.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so, the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space is occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

10. State the uses of pointers.

- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

11. What is the difference between the address stored in a pointer and a value at the address?

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address.

12. What is the difference between the indirection operator and the address of operator?

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

13. State the difference between call by value and call by reference.

Call by value	Call by reference
Formal parameter is a local variable.	Formal parameter is a reference variable.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

14. State the advantages of using bit-fields.

- To store Boolean variables (true/false) in one bit.
- To encrypt certain routines to access the bits within a byte.
- To transmit status information of devices encoded into one or more bits within a byte.

15. State the difference between arrays and structures.

Arrays	Structures
An array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.
Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we are dealing with structures.

Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are stored in sequence of memory locations.
The array elements are accessed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.

Its general format:
 	
Data type array name [size];
	Its general format is:
	struct structure name
	{
	    data_type structure member 1;
	    data_type structure member 2;
	          .
	          .
	          .
	    data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	    char studname [25];
	    int rollno;
	} stud1;

16. State the difference between structures and unions.

Structures	Unions
Each member in a structure occupies and uses its own memory space.	All the members of a union use the same memory space.
The keyword struct tells us that we are dealing with structures.	The keyword union tells us that we are dealing with unions.
All the members of a structure can be initialized.	Only one member of a union can be initialized.
More memory space is required since each member is stored in a separate memory locations.	Less memory space is required since all members are stored in the same memory locations.

Its general format is:
struct structure name
{
   data_type structure Member1;
   data_type structure Member2;
       .
       .
   data_type structure Member N;
} structure variable;
	Its general formula is:
	union union name
	{
	   data_type structure Member1;
	   data_type structure Member2;
	       .
	       .
	   data_type structure Member N;
	} union variable;

Example:
struct student
{
    char studname[25];
    int rollno;
} stud1;
	Example:
	union student
	{
	    char studname[25];
	    int rollno;
	} stud1;

17. Advantages of functions:

   1. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

   2. The length of source program can be reduced by using function by using it at different places in the program according to the user's requirement.

   3. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

   4. A function can be used by many programs.

   5. By using the function, portability of the program is very easy.

   6. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

   7. Debugging (removing error) becomes very easier and fast using the function sub-programming.

   8. Functions are more flexible than library functions.

   9. Testing (verification and validation) is very easy by using functions.

   10. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

18. A complete summarized table to represent the lifetime and visibility (scope) of all the
    storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local

19. Tokens:

   Tokens are the smallest individual unit in a program. The different types of tokens used in C are as follows:

   • Keywords.
   • Identifiers.
   • Constants.
   • Operators.
   • Strings.

SAMPLE C – PROGRAMS

1.   Write a C programme to accept a list of data items & find the II largest & II smallest in it & take average of both & search for that value. Display appropriate message on successful search.

```
main ()
{
        int i,j,a,n,counter,ave,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
        scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                for (j=i+1; j<n; ++j)
                {
                        if (number[i] < number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        printf ("The numbers arranged in ascending order are given below\n");
        for (i=0; i<n; ++i)
                printf ("%10d\n",number[i]);
        printf ("The 2nd largest number is  = %d\n", number[1]);
        printf ("The 2nd smallest number is = %d\n", number[n-2]);
        ave = (number[1] +number[n-2])/2;
        counter = 0;
        for (i=0; i<n; ++i)
        {
                if (ave==number[i])
                ++counter;
        }
        if (counter==0)
                printf("The average of 2nd largest & 2nd smallest is not in the array\n");
        else
                printf("The average of 2nd largest & 2nd smallest in array is %d in
                        numbers\n", counter);
}
```

2.   Write a C programme to arrange the given numbers in ascending order.

```c
main ()
{
        int i,j,a,n,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] > number[j])
                                {
                                        a= number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
        }
        printf("Number in Asscending order:\n");
        for(i=0;i<n;i++)
                printf("\t%d\n",number[i]);
}
```

3.   Program to convert the given binary number into decimal.

```c
# include <stdio.h>
main()
{
        int   num, bnum, dec = 0, base = 1, rem ;
        printf("Enter the binary number(1s and 0s)\n");
        scanf("%d", &num);          /*maximum five digits */
        bnum = num;
        while( num > 0)
        {
                        rem = num % 10;
                        if(rem>1)
                        {
                                printf("\nError in input");
                                break;
                        }
                        dec = dec + rem * base;
                        num = num / 10 ;
                        base = base * 2;
        }
        if(num==0)
        {
                printf("The Binary number is = %d\n", bnum);
                printf("Its decimal equivalent is =%d\n", dec);
        }
}
```

4.   Write to accept a 1-dimensional array of N elements & split into 2 halves & sort 1st half in ascending order & 2nd into descending order.

```c
#include<stdio.h>

main ()
{
        int i,j,a,n,b,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        b = n/2;
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<b; ++i)
        {
                for (j=i+1; j<b; ++j)
                {
                        if (number[i] > number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        for (i=b; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] < number[j])
                                {
                                        a = number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
         }
        printf (" The 1st half numbers\n");
        printf (" arranged in asc\n");
        for (i=0; i<b; ++i)
                printf ("%d ",number[i]);
        printf("\nThe 2nd half Numbers\n");
        printf("order arranged in desc.order\n");
        for(i=b;i<n;i++)
                printf("%d ",number[i]);
}
```

5. Program to delete the desired element from the list.

```c
# include <stdio.h>

main()
{
        int  vectx[10];
        int  i, n, found = 0, pos, element;

        printf("Enter how many elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &vectx[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", vectx[i]);
        }

         printf("Enter the element to be deleted\n");
         scanf("%d",&element);

        for(i=0; i<n; i++)
        {
                if ( vectx[i] == element)
                {
                        found = 1;
                        pos = i;
                        break;
                }
        }
        if (found == 1)
        {
                for(i=pos; i< n-1; i++)
                {
                        vectx[i] = vectx[i+1];
                }
                printf("The resultant vector is \n");
                for(i=0; i<n-1; i++)
                {
                        printf("%d\n",vectx[i]);
                }
        }
        else
                printf("Element %d is not found in the vector\n", element);

}
```

6.   Write a "C" program to Interchange the main diagonal elements with the scondary diagonal elements.

```c
#include<stdio.h>
main ()
{
      int i,j,m,n,a;
      static int ma[10][10];
      printf ("Enetr the order of the matix \n");
      scanf ("%dx%d",&m,&n);

      if (m==n)
      {
            printf ("Enter the co-efficients of the matrix\n");
            for (i=0;i<m;++i)
            {
                  for (j=0;j<n;++j)
                  {
                        scanf ("%dx%d",&ma[i][j]);
                  }
            }
            printf ("The given matrix is \n");
            for (i=0;i<m;++i)
            {
                  for (j=0;j<n;++j)
                  {
                        printf (" %d",ma[i][j]);
                  }
                  printf ("\n");
            }
            for (i=0;i<m;++i)
            {
                  a = ma[i][i];
                  ma[i][i]   = ma[i][m-i-1];
                  ma[i][m-i-1] = a;
            }
            printf ("THe matrix after changing the \n");
            printf ("main diagonal & secondary diagonal\n");
            for (i=0;i<m;++i)
            {
                  for (j=0;j<n;++j)
                  {
                        printf (" %d",ma[i][j]);
                  }
                  printf ("\n");
            }
      }
      else
            printf ("The given order is not square matrix\n");
}
```

7.   Program to insert an element at an appropriate position in an array.

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

main()
{
        int  x[10];
        int  i, j, n, m, temp, key, pos;

        clrscr();
        printf("Enter how many elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &x[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }

        for(i=0; i< n; i++)
        {
                for(j=i+1; j<n; j++)
                {
                        if (x[i] > x[j])
                        {
                                temp = x[i];
                                x[i] = x[j];
                                x[j] = temp;
                        }
                }
        }
        printf("Sorted list is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }

        printf("Enter the element to be inserted\n");
        scanf("%d",&key);

        for(i=0; i<n; i++)
        {
                if ( key < x[i] )
                {
                        pos = i;
                        break;
                }
        }
        m = n - pos + 1 ;
        for(i=0; i<= m ; i++)
        {
```

```c
                x[n-i+2] = x[n-i+1] ;
        }
        x[pos] = key;

        printf("Final list is:\n");
        for(i=0; i<n+1; i++)
        {
                printf("%d\n", x[i]);
        }
}
```

8.   Program to generate the fibonacci sequence.

```c
#include <stdio.h>
main()
{
        int   fib1=0, fib2=1, fib3, limit, count=0;

        printf("Enter the limit to generate the fibonacci sequence\n");
        scanf("%d", &limit);

        printf("Fibonacci sequence is ...\n");
        printf("%d\n",fib1);
        printf("%d\n",fib2);
        count = 2;                      /* fib1 and fib2 are already used */

        while( count < limit)
        {
                fib3 = fib1 + fib2;
                count ++;
                printf("%d\n",fib3);
                fib1 = fib2;
                fib2 = fib3;
        }
}
```

9.   Program to compute mean, varience and standard deviation.

```c
main()
{
                float x[10];
                int  i, n;
                float avrg, var, SD, sum=0, sum1=0;

                printf("Enter how many elements\n");
                scanf("%d", &n);
                printf("Enter %d numbers:",n);
                for(i=0; i<n; i++)
                {
                        scanf("%f", &x[i]);
                }
                                        /* Compute the sum of all elements */
                for(i=0; i<n; i++)
                {
                        sum = sum + x[i];
```

```c
        }
        avrg = sum /(float) n;
                                /* Compute varaience and standard deviation  */
        for(i=0; i<n; i++)
        {
                sum1 = sum1 + pow((x[i] - avrg),2);
        }
        var = sum1 / (float) n;
        SD = sqrt(var);

        printf("Average of all elements =%.2f\n", avrg);
        printf("Varience of all elements =%.2f\n", avrg);
        printf("Standard Deviation of all elements =%.2f\n", avrg);
}
```

10.   Program to sort names in alphabetical order using structures.

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

struct tag
{
        char name[10];
        int rno;
};

typedef struct tag node;

node  s[5];

sort(int no)
{
        int i,j;
        node temp;
        for(i=0;i<no-1;i++)
                for(j=i+1;j<no;j++)
                        if(strcmp(s[i].name,s[j].name)>0)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
}


main()
{
        int no,i;
        clrscr();
        fflush(stdin);

        printf("Enter The Number Of Students:");
        scanf("%d",&no);

        for(i=0;i<no;i++)
```

```c
{
        printf("Enter The Name:");
        fflush(stdin);
        gets(s[i].name);
        printf("Enter the Roll:");
        scanf("%d",&s[i].rno);
}

sort(no);
for(i=0;i<no;i++)
{
        printf("%s\t",s[i].name);
        printf("%d\n",s[i].rno);
}
getche();
}
```

11.   Program to reverse the given integer (palindrome).

```c
#include <stdio.h>
main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);

        if(temp == rev )
                printf("Number is a palindrome\n");
        else
                printf("Number is not a palindrome\n");
}
```

12.   Program to find the frequency of odd numbers & even numbers in the input of a
       matrix.
```c
#include<stdio.h>
main ()
{
        int i,j,m,n,even=0,odd=0;
        static int ma[10][10];
        printf ("Enter the order ofthe matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefficients if matrix \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
```

```c
        {
                scanf ("%d", &ma[i][j]);
                if ((ma[i][j]%2) == 0)
                {
                        ++even;
                }
                else
                        ++odd;
        }
}
printf ("The given matrix is\n");
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
                printf (" %d",ma[i][j]);
        printf ("\n");
}
printf ("The frequency of odd number occurance = %d\n",odd);
printf ("The frequency of even number occurance = %d\n",even);
}
```

13.   Program to determine the given number is odd.

```c
#include <stdio.h>

main()
{
        int numb;
        printf(" Enter the number\n");
        scanf("%d", &numb);

        if((numb%2)!=0)
                printf(" %d , is an odd number\n", numb);

}
```

14.   Program to check if the two numbers are equal.

```c
#include <stdio.h>

main()
{
        int m,n;
        printf(" Enter the values for  M and N\n");
        scanf("%d%d", &m,&n);
        if(m == n )
                printf("M and  N are equal\n");
        else
                printf("M and N are not equal\n");
}
```

15. Program to find the largest among three numbers.

```c
#include <stdio.h>

main()
{
        int a,b,c;
        printf(" Enter  the values for A,B,C\n");
        scanf("%d %d %d", &a,&b,&c);
        if( a > b )
        {
                if ( a > c)
                        printf(" A is the Largest\n");
                else
                        printf("C is the largest\n");
        }
        else if ( b > c)
                printf(" B is the Largest\n");
        else
                printf("C is the Largest\n");
}
```

16. Program to determine the quadrant of a point in a cartesian coordinate system.

```c
#include <stdio.h>
main()
{
        int x,y;
        printf("Enter the values for X and Y\n");
        scanf("%d %d",&x,&y);

        if( x > 0 && y > 0)
                printf(" point (%d,%d) lies in the First quandrant\n",x,y);
        else if( x < 0 && y > 0)
                printf(" point (%d,%d) lies in the Second quandrant\n",x,y);
        else if( x < 0 && y < 0)
                printf(" point (%d, %d) lies in the Third quandrant\n",x,y);
        else if( x > 0 && y < 0)
                printf(" point (%d,%d) lies in the Fourth quandrant\n",x,y);
        else if( x == 0 && y == 0)
                printf(" point (%d,%d) lies at the origin\n",x,y);
}
```

17. Program to find the areas of different geometrical figures using switch statement.

```c
#include <stdio.h>
main()
{
        int  fig_code;
        float  side,base,length,bredth,height,area,radius;
        printf("-----------------------\n");
        printf(" 1 --> Circle\n");
        printf(" 2 --> Rectangle\n");
        printf(" 3 --> Triangle\n");
        printf(" 4 --> Square\n");
```

```c
        printf("-----------------------\n");

        printf("Enter the Figure code\n");
        scanf("%d",&fig_code);

        switch(fig_code)
        {
                case 1:
                        printf(" Enter the radius\n");
                        scanf("%f",&radius);
                        area=3.142*radius*radius;
                        printf("Area of a circle=%f\n", area);
                        break;
                case 2:
                        printf(" Enter the bredth and length\n");
                        scanf("%f %f",&bredth, &length);
                        area=bredth *length;
                        printf("Area of a Reactangle=%f\n", area);
                        break;
                case 3:
                        printf(" Enter the base and height\n");
                        scanf("%f %f",&base,&height);
                        area=0.5 *base*height;
                        printf("Area of a Triangle=%f\n", area);
                        break;
                case 4:
                        printf(" Enter the side\n");
                        scanf("%f",&side);
                        area=side * side;
                        printf("Area of a Square=%f\n", area);
                        break;
                default:
                        printf(" Error in figure code\n");
                        break;
        }                               /* End of switch */

}               /* End of main() */
```

18.   Program to grade the student.

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

main()
{
        char remark[15], grade;

        printf("Enter the grade\n");
        scanf("%c",&grade);

        grade=toupper(grade);                   /* lower case letter to upper case */

        switch(grade)
        {
                case 'S':
                        strcpy(remark," SUPER");
```

```c
                break;
        case 'A':
                strcpy(remark," VERY GOOD");
                break;
        case 'B':
                strcpy(remark," FAIR");
                break;
        case 'Y':
                strcpy(remark," ABSENT");
                break;
        case 'F':
                strcpy(remark," FAILS");
                break;
        default :
                strcpy(remark, "ERROR IN GRADE\n");
                break;
        }                                       /* End of switch */
        printf("RESULT: %s\n",remark);
}
```

19.   Program to find the sum of first 50 natural numbers.

```c
#include <stdio.h>
main()
{
        int  num,sum=0;
        for(num=1;num<=50; num++)
                sum += num;
        printf("Sum =%4d\n", sum);
}
```

20.   Program to illustrate compound statement with for loop.

```c
#include <stdio.h>
main()
{
        int  index=0, value=0,number;
        printf("Enter The Number\n");
        scanf("%d", &number);
        for(index=1;index<=number;index++)
        {
                value++;
                printf("%d\n",value);
        }
        printf("Goodbye\n");
}
```

21.   Program to find the factorial of a number.

```c
#include <stdio.h>
main()
{
        int  i,fact=1,num;
        printf("Enter the number\n");
```

```c
        scanf("%d",&num);
        if( num <0)
                printf("Factorial is not there for -ve numbers");
        else if(num==0 || num==1)
                fact=1;
        else
        {
                for(i=1;i<=num; i++)
                        fact *= i;
        }
        printf(" Factorial of %d =%5d\n", num,fact);
}
```

22. Program to illustrate for loop without initial and increment/decrement expressions.

```c
#include <stdio.h>
main()
{
        int  i=0,limit=5;
        printf(" Values of I\n");
        for(  ; i<limit;   )
        {
                i++;
                printf("%d\n",i);
        }
}
```

23.   Program to accept a string and find the sum of all digits in the string.

```c
#include <stdio.h>
main()
{
        char string[80];
        int count,nc=0,sum=0;
        printf("Enter the string containing both digits and alphabet\n");
        scanf("%s",string);
        for(count=0;string[count]!='\0'; count++)
        {
                if((string[count]>='0') && (string[count]<='9'))
                {
                        nc += 1;
                        sum += (string[count] - '0');
                }
        }
        printf("NO. of Digits in the string=%d\n",nc);
        printf("Sum of all digits=%d\n",sum);

}
```

24.   Program to find the sum of the sine series.

```c
#include <stdio.h>
#include <math.h>
#define pi 3.142
```

```c
main()
{
        int  i,n,k,sign;
        float sum=0,num,den,xdeg,xrad,xsqr,term;
        printf("Enter the angle( in degree)\n");
        scanf("%f",&xdeg);
        printf("Enter the no. of terms\n");
        scanf("%d",&n);
        xrad=xdeg * (pi/180.0);    /* Degrees to radians*/
        xsqr= xrad*xrad;
        sign=1; k=2; num=xrad; den=1;

        for(i=1;i<=n; i++)
        {
                term=(num/den)* sign;
                sum += term;
                sign *= -1;
                num *= xsqr;
                den *= k*(k+1);
                k += 2;
        }
        printf("Sum of sine series of %d terms =%8.3f\n",n,sum);
}
```

25.   Program to find the sum of cos(x) series.

```c
#include<stdio.h>
#include<math.h>
main()
{
        float x,sign,cosx,fact;
        int n,x1,i,j;
        printf("Enter the number of the terms in a series\n");
        scanf("%d",&n);
        printf("Enter the value of x(in degrees)\n");
        scanf("%f",&x);
        x1=x;
        x=x*(3.142/180.0); /* Degrees to radians*/
        cosx=1;
        sign=-1;
        for(i=2;i<=n;i=i+2)
        {
                fact=1;
                for(j=1;j<=i;j++)
                {
                        fact=fact*j;
                }
                cosx=cosx+(pow(x,i)/fact)*sign;
                sign=sign*(-1);
        }
        printf("Sum of the cosine series=%f\n",cosx);
        printf("The value of cos(%d) using library function=%f\n",x1,cos(x));
}
```

26.   Program to find the sum of all elements of an array using pointers.

```c
#include <stdio.h>

main()
{
        static int array[5]={ 200,400,600,800,1000 };
        int addnum(int *ptr);     /* function prototype */

        int sum;
        sum = addnum(array);
        printf(" Sum of all array elements=%d\n", sum);

}

int  addnum(int *ptr)
{
        int total=0, index;
        for(index=0;index<5; index++)
                total+=*(ptr+index);
        return(total);
}
```

27.   Program to swap particular elements in array using pointers.

```c
#include <stdio.h>

main()
{
        float x[10];
        int i,n;
        float swap34(float *ptr1, float  *ptr2 );     /* Function Declaration */
        printf(" How many Elements...\n");
        scanf("%d", &n);
        printf(" Enter Elements one by one\n");
        for(i=0;i<n;i++)
                scanf("%f",x+i);

        swap34(x+2, x+3);                          /* Interchanging 3rd element by 4th */
        printf(" Resultant Array...\n");
        for(i=0;i<n;i++)
                printf("X[%d]=%f\n",i,x[i]);
}

float swap34(float *ptr1, float *ptr2 )             /* Function Definition */
{
        float temp;
        temp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=temp;
}
```

28.   Program to find the sum of two 1-D arrays using Dynamic Memory Allocation.

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

main()
{
        int i,n,sum;
        int *a,*b,*c;
        printf(" How many Elements in each array...\n");
        scanf("%d", &n);
        a=(int *) malloc(sizeof(n*(int)));
        b=(int *) malloc(sizeof(n*(int)));
        c=(int *) malloc(sizeof(n*(int)));
        printf(" Enter Elements of First List\n");
        for(i=0;i<n;i++)
                scanf("%f",a+i);

        printf(" Enter Elements of Second List\n");
        for(i=0;i<n;i++)
                scanf("%f",b+i);

        for(i=0;i<n;i++)
                c+i=(a+i) +( b+i);
        printf(" Resultant List is\n");
        for(i=0;i<n;i++)
                printf("C[%d]=%f\n",i,c[i]);
}
```

29.   Program to reverse the given integer.

```c
#include <stdio.h>

main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);
}
```

30.   Program to sort all rows of matrix in ascending order & all columns in descending order.

```
main ()
{
    int i,j,k,a,m,n;
    static int ma[10][10],mb[10][10];
    printf ("Enter the order of the matrix \n");
    scanf ("%d %d", &m,&n);
    printf ("Enter co-efficients of the matrix \n");
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            scanf ("%d",&ma[i][j]);
            mb[i][j] = ma[i][j];
        }
    }
    printf ("The given matrix is \n");
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            printf (" %d",ma[i][j]);
        }
        printf ("\n");
    }
    printf ("After arranging rows in ascending order\n");
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            for (k=(j+1);k<n;++k)
            {
                if (ma[i][j] > ma[i][k])
                {
                    a = ma[i][j];
                    ma[i][j] = ma[i][k];
                    ma[i][k] = a;
                }
            }
        }
    }
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            printf (" %d",ma[i][j]);
        }
        printf ("\n");
    }
    printf ("After arranging the columns in descending order \n");
    for (j=0;j<n;++j)
    {
        for (i=0;i<m;++i)
        {
            for (k=i+1;k<m;++k)
```

```
                        {
                                if (mb[i][j] < mb[k][j])
                                {
                                        a = mb[i][j];
                                        mb[i][j] = mb[k][j];
                                        mb[k][j] = a;
                                }
                        }
                }
        }
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",mb[i][j]);
                }
                printf ("\n");
        }
}
```

31.   Program to convert lower case letters to upper case vice-versa.

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

main()
{
        char sentence[100];
        int count, ch, i;
        clrscr();

        printf("Enter a sentence\n");
        for(i=0; (sentence[i] = getchar())!='\n'; i++);
        count = i;
        sentence[count]='\0';
        printf("Input sentence is : %s",sentence);
        printf("\nResultant sentence is\n");
        for(i=0; i < count; i++)
        {
                        ch = islower(sentence[i]) ? toupper(sentence[i]) :
tolower(sentence[i]);
                        putchar(ch);
                }
}
```

32.   Program to find the sum of the rows & columns of a matrix.

```
main ()
{
        int i,j,m,n,sum=0;
        static int m1[10][10];
        printf ("Enter the order of the matrix\n");
        scanf ("%d%d", &m,&n);
        printf ("Enter the co-efficients of the matrix\n");
        for (i=0;i<m;++i)
```

```
{
        for (j=0;j<n;++j)
        {
                scanf ("%d",&m1[i][j]);
        }
}
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
                sum = sum + m1[i][j] ;
        }
        printf ("   Sum of the %d row is = %d\n",i,sum);
        sum = 0;
}
sum=0;
for (j=0;j<n;++j)
{
        for (i=0;i<m;++i)
        {
                sum = sum+m1[i][j];
        }
         printf ("Sum of the %d column is = %d\n", j,sum);
        sum = 0;
}
}
```

33.   Program to find the transpose of a matrix.

```
#include<stdio.h>
main ()
{
        int i,j,m,n;
        static int ma[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefiicients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
        printf ("Transpose of matrix is \n");
        for (j=0;j<n;++j)
        {
```

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

34.   Program to illustrate the unions.

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

main()
{
        union number
        {
                int  n1;
                float n2;
        };
        union number x;
        clrscr() ;
        printf("Enter the value of n1: ");
        scanf("%d", &x.n1);
        printf("Value of n1 =%d", x.n1);
        printf("\nEnter the value of n2: ");
        scanf("%d", &x.n2);
        printf("Value of n2 = %d\n",x.n2);
}
```

35.   Program to accepts two strings and compare them. Finally print whether, both
are equal, or first string is greater than the second or the first string is less than the
second string without using string library.

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

void main()
{
        int count1=0,count2=0,flag=0,i;
        char str1[10],str2[10];

        clrscr();
        puts("Enter a string:");
        gets(str1);

        puts("Enter another string:");
        gets(str2);
                                    /*Count the number of characters in str1*/
        while (str1[count1]!='\0')
                count1++;
                                    /*Count the number of characters in str2*/
        while (str2[count2]!='\0')
                count2++;
        i=0;
```

/*The string comparison starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until the end of the strings is reached.*/

```c
        while ( (i < count1) && (i < count2))
        {
                if (str1[i] == str2[i])
                {
                        i++;
                        continue;
                }
                if (str1[i]<str2[i])
                {
                        flag = -1;
                        break;
                }
                if (str1[i] > str2[i])
                {
                        flag = 1;
                        break;
                }
        }

        if (flag==0)
            printf("Both strings are equal\n");
        if (flag==1)
            printf("String1 is greater than string2\n", str1, str2);
        if (flag == -1)
            printf("String1 is less than string2\n", str1, str2);
        getch();
}
```

Output:

Enter a string: happy
Enter another string: HAPPY
String1 is greater than string2

RUN2:
Enter a string: Hello
Enter another string: Hello
Both strings are equal

RUN3:
Enter a string: gold
Enter another string: silver
String1 is less than string2

36.   Program to accept a decimal number and convert it binary and count the number of 1's in the binary number.

```c
#include <stdio.h>

main()
{
```

```c
        long   num, dnum, bin = 0, base = 1;
        int rem, no_of_1s = 0 ;

        printf("Enter a decimal integer:\n");
        scanf("%ld", &num);                              /*maximum five digits */
        dnum = num;

        while( num > 0)
        {
                rem = num % 2;
                if (rem==1)                    /*To count no.of 1s*/
                {
                        no_of_1s++;
                }
                bin = bin + rem * base;
                num = num / 2 ;
                base = base * 10;
        }
        printf("Input number is = %ld\n", dnum);
        printf("Its Binary equivalent is =%ld\n", bin);
        printf("No. of 1's in the binary number is = %d\n", no_of_1s);
}
```

Output:

Enter a decimal integer: 75
Input number is = 75
Its Binary equivalent is =1001011
No. of 1's in the binary number is = 4

RUN2
Enter a decimal integer: 128
Input number is = 128
Its Binary equivalent is=10000000
No. of 1's in the binary number is = 1

37.   Program to calculate the salary of an employee.

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

void main()
{
        struct employee
        {
                long int code;
                char name[20];
                char vertical[15];
                float salary;
        };
        struct employee emp;
        clrscr();
        printf("employee code:");
        scanf("%ld",&emp.code);
        fflush(stdin);
```

```c
        printf("employee name:");
        scanf("%[^\n]",emp.name);
        fflush(stdin);
        printf("vertical:");
        scanf("%[^\n]",emp.vertical);
        fflush(stdin);
        printf("salary:");
        scanf("%f",&emp.salary);
        clrscr();
        printf("      EMPLOYEE INFORMATION SYSTEM\n");
        printf("------------------------------------------------\n");
        printf("CODE |    NAME    |   VERTICAL   |SALARY\n");
        printf("%ld | %-15s| %-15.10s| 7.2f\n", emp.code, emp.name, emp.vertical,
                emp.salary);
        printf("-----------------------------------------------\n");
        getch();
}
```

38.   Program to accept N integer number and store them in an array AR. The odd elements in the AR are copied into OAR and other elements are copied into EAR. Display the contents of OAR and EAR.

```c
 #include <stdio.h>

main()
 {
        long int ARR[10], OAR[10], EAR[10];
        int i,j=0,k=0,n;

        printf("Enter the size of array AR:\n");
        scanf("%d",&n);

        printf("Enter the elements of the array:\n");
        for(i=0;i<n;i++)
        {
                scanf("%ld",&ARR[i]);
                fflush(stdin);
        }
                        /*Copy odd and even elemets into their respective arrays*/
        for(i=0;i<n;i++)
        {
                if (ARR[i]%2 == 0)
                {
                        EAR[j] = ARR[i];
                        j++;
                }
                else
                {
                        OAR[k] = ARR[i];
                        k++;
                }
        }

        printf("The elements of OAR are:\n");
        for(i=0;i<j;i++)
        {
                printf("%ld\n",OAR[i]);
```

```
        }

        printf("The elements of EAR are:\n");
        for(i=0;i<k;i++)
        {
                printf("%ld\n", EAR[i]);
        }
}
```

Output:

```
Enter the size of array AR: 6
Enter the elements of the array:
12
345
678
899
900
111

The elements of OAR are:
 345
 899
 111

The elements of EAR are:
 12
 678
 900
```

39.

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

void main()
{      struct date
       {
               int day;
               int month;
               int year;
       };
       struct details
       {
               char name[20];
               int price;
               int code;
               int qty;
               struct date mfg;
       };

       struct details item[50];
       int n,i;

       clrscr();
       printf("Enter number of items:");
```

230

```c
        scanf("%d",&n);
        fflush(stdin);
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                printf("Item name:");
                scanf("%[^\n]",item[i].name);

                fflush(stdin);
                printf("Item code:");
                scanf("%d",&item[i].code);
                fflush(stdin);

                printf("Quantity:");
                scanf("%d",&item[i].qty);
                fflush(stdin);

                printf("price:");
                scanf("%d",&item[i].price);
                fflush(stdin);

                printf("Manufacturing date(dd-mm-yyyy):");
                scanf("%d-%d-%d",&item[i].mfg.day,&item[i].mfg.month,&item[i].mfg.year);
        }
        printf("          *****  INVENTORY *****\n");
        printf("-------------------------------------------------------------------\n");
        printf("S.N.|   NAME          |  CODE  |  QUANTITY | PRICE |MFG.DATE\n");
        printf("-------------------------------------------------------------\n");
        for(i=0;i<n;i++)
                printf("%d     %-15s      %-d        %-5d    %-5d
                        %d/%d/%d\n",i+1,item[i].name,item[i].code,item[i].qty,item[i].pr
                        ice,item[i].mfg.day,item[i].mfg.month,item[i].mfg.year);
        printf("-------------------------------------------------------------\n");
        getch();
}
```

40.   Program to find whether a given year is leap year or not.

```c
 # include <stdio.h>

 main()
 {
        int year;

        printf("Enter a year:\n");
        scanf("%d",&year);

        if ( (year % 4) == 0)
                printf("%d is a leap year",year);
        else
                printf("%d is not a leap year\n",year);
 }
```

Output:

Enter a year: 2000

2000 is a leap year

RUN2:

Enter a year: 1999
1999 is not a leap year

41.   Program to multiply given number by 4 using bitwise operators.

```
# include <stdio.h>

main()
{
    long number, tempnum;
    printf("Enter an integer:\n");
    scanf("%ld",&number);
    tempnum = number;
    number = number << 2;   /*left shift by two bits*/

    printf("%ld x 4 = %ld\n", tempnum,number);
}
```

Output:

Enter an integer: 15
15 x 4 = 60

RUN2:
Enter an integer: 262
262 x 4 = 1048

42.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int count1=0,count2=0,i,j,flag;
    char str[80],search[10];
    clrscr();
    puts("Enter a string:");
    gets(str);
    puts("Enter search substring:");
    gets(search);
    while (str[count1]!='\0')
            count1++;
    while (search[count2]!='\0')
            count2++;
    for(i=0;i<=count1-count2;i++)
    {
            for(j=i;j<i+count2;j++)
            {
                    flag=1;
                    if (str[j]!=search[j-i])
                    {
```

```
                              flag=0;
                              break;
                        }
                  }
                  if (flag==1)
                        break;
            }
      if (flag==1)
            puts("SEARCH SUCCESSFUL!");
      else
            puts("SEARCH UNSUCCESSFUL!");
      getch();
}
```

43.   Program to compute the value of X ^ N given X and N as inputs.

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

void main()
{
      long int x,n,xpown;
      long int power(int x, int n);

      printf("Enter the values of X and N: \n");
      scanf("%ld %ld",&x,&n);

      xpown = power (x,n);

      printf("X to the power N = %ld\n");
}

/*Recursive function to computer the X to power N*/

long int power(int x, int n)
{
      if (n==1)
            return(x);
       else if ( n%2 == 0)
            return (pow(power(x,n/2),2));  /*if n is even*/
      else
            return (x*power(x, n-1));    /* if n is odd*/
}
```

Output:

Enter the values of X and N: 2 5
X to the power N = 32

RUN2:
Enter the values of X and N: 4 4
X to the power N ==256

RUN3:
Enter the values of X and N: 5 2
X to the power N = 25

233

RUN4:
Enter the values of X and N: 10 5
X to the power N = 100000

44.   Program to accept a string and find the number of times the word 'the' appears in it.

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

void main()
{
        int count=0,i,times=0,t,h,e,space;
        char str[100];

        clrscr();
        puts("Enter a string:");
        gets(str);
                                /*Traverse the string to count the number of characters*/
        while (str[count]!='\0')
        {
                count++;
        }
                                /*Finding the frequency of the word 'the'*/
        for(i=0;i<=count-3;i++)
        {
                t=(str[i]=='t'||str[i]=='T');
                h=(str[i+1]=='h'||str[i+1]=='H');
                e=(str[i+2]=='e'||str[i+2]=='E');
                space=(str[i+3]==' '||str[i+3]=='\0');
                if ((t&&h&&e&&space)==1)
                   times++;
        }
        printf("Frequency of the word \'the\' is %d\n",times);
        getch();
}
```

Output:

Enter a string: The Teacher's day is the birth day of Dr.S.Radhakrishnan
Frequency of the word 'the' is 2

45.   Program to find the number of characters, words and lines.

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

void main()
{
        int count=0,chars,words=0,lines,i;
        char text[1000];

        clrscr();
```

```
        puts("Enter text:");
        gets(text);

        while (text[count]!='\0')
                count++;
        chars=count;

        for (i=0;i<=count;i++)
        {
                if ((text[i]==' '&&text[i+1]!=' ')||text[i]=='\0')
                    words++;
        }

        lines=chars/80+1;
        printf("no. of characters:%d\n",chars);
        printf("no. of words:%d\n",words);
        printf("no. of lines:%d\n",lines);
        getch();
}
```

46.   Program to swap the contents of two numbers using bitwise XOR operation. Don't use either the temporary variable or arithmetic operators.

```
# include <stdio.h>

main()
{
    long i,k;
    printf("Enter two integers: \n");
    scanf("%ld %ld",&i,&k);
    printf("\nBefore swapping i= %ld and k = %ld",i,k);
    i = i^k;
    k = i^k;
    i = i^k;
    printf("\nAfter swapping i= %ld and k = %ld",i,k);

}
```

Output:

Enter two integers: 23 34
Before swapping i= 23 and k = 34
After swapping i= 34 and k = 23

47.   Program to find the length of a string without using the built-in function, also check whether it is a palindrome or not.

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

main()
{
    char string[25], revString[25]={'\0'};
    int  i, length = 0, flag = 1;
```

```c
        clrscr();
        fflush(stdin);

        printf("Enter a string\n");
        gets(string);

        for (i=0; string[i] != '\0'; i++)      /*keep going through each */
        {                                       /*character of the string */
                length++;                        /*till its end */
        }
        printf("The length of the string: \'%s\' = %d\n", string, length);

        for (i=length-1; i >= 0 ; i--)
        {
                revString[length-i-1] = string[i];
        }
                /*Compare the input string and its reverse. If both are equal
         then the input string is palindrome. Otherwise it is not a palindrome */

        for (i=0; i < length ; i++)
        {
                if (revString[i] != string[i])
                        flag = 0;
        }

        if (flag == 1)
                printf ("%s is a palindrome\n", string);
        else
                printf("%s is not a palindrome\n", string);
}
```

Output:

```
Enter a string: madam
The length of the string 'madam' = 5
madam is a palindrome

RUN2:
Enter a string: good
The length of the string 'good' = 4
good is not a palindrome
```

48.   Program to accept two strings and concatenate them i.e. the second string is appended to the end of the first string.

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

main()
{
    char string1[20], string2[20];
    int i,j,pos;

    strset(string1, '\0');                /*set all occurrences in two strings to NULL*/
    strset(string2,'\0');
```

```c
    printf("Enter the first string:");
    gets(string1);
    fflush(stdin);
    printf("Enter the second string:");
    gets(string2);

    printf("First string  = %s\n", string1);
    printf("Second string = %s\n", string2);

                            /*To concate the second stribg to the end of the string
    travserse the first to its end and attach the second string*/
    for (i=0; string1[i] != '\0'; i++)
    {
       ;                    /*null statement: simply trvsering the string1*/
    }

    pos = i;
    for (i=pos,j=0; string2[j]!='\0'; i++)
    {
        string1[i] = string2[j++];
    }
    string1[i]='\0';            /*set the last character of string1 to NULL*/

    printf("Concatenated string = %s\n", string1);
}
```

Output:

```
Enter the first string: CD-
Enter the second string: ROM
First string  = CD-
Second string = ROM
Concatenated string = CD-ROM
```

49.

```c
#include <stdio.h>
 int A[10][10], B[10][10], sumat[10][10], diffmat[10][10];
 int i, j, R1, C1, R2, C2;
main()
{
        void readmatA();
        void printmatA();
        void readmatB();
        void printmatB();
        void sum();
        void diff();

        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);

        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
```

```c
        if( R1 != R2 && C1 != C2)
        {
                printf("Addition and subtraction are possible\n");
                exit(1);
        }
        else
        {
                printf("Enter the elements of matrix A\n");
                readmatA();
                printf("MATRIX A is\n");
                printmatA();

                printf("Enter the elements of matrix B\n");
                readmatB();
                printf("MATRIX B is\n");
                printmatB();

                sum();
                diff();
        }
        return 0;
}

void readmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        return;
}

void readmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
}

void printmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
}
```

```c
void printmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
}

void sum()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        sumat[i][j] = A[i][j] + B[i][j];
                }
        }
        printf("Sum matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",sumat[i][j]) ;
                }
                printf("\n");
        }
        return;
}

void diff()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        diffmat[i][j] = A[i][j] - B[i][j];
                }
        }
        printf("Difference matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",diffmat[i][j]);
                }
                printf("\n");
        }
        return;
}
```

50.

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

main()
{
        int A[10][10], B[10][10];
        int i, j, R1, C1, R2, C2, flag =1;

        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);

        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        printf("Enter the elements of matrix A\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        printf("Enter the elements of matrix B\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
        printf("MATRIX A is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
        printf("MATRIX B is\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
                        /* Comparing two matrices for equality */

        if(R1 == R2 && C1 == C2)
        {
                printf("Matrices can be compared\n");
                for(i=0; i<R1; i++)
                {
                        for(j=0; j<C2; j++)
```

```
                        {
                                if(A[i][j] != B[i][j])
                                {
                                        flag = 0;
                                        break;
                                }
                        }
                }
        }
        else
        {
                printf(" Cannot be compared\n");
                exit(1);
        }

        if(flag == 1 )
                printf("Two matrices are equal\n");
        else
                printf("But,two matrices are not equal\n");
}
```

51.   Program to illustrate how user authentication is made before allowing the user to access the secured resources. It asks for the user name and then the password. The password that you enter will not be displayed, instead that character is replaced by '*'.

```
#include <stdio.h>

void main()
{
        char pasword[10],usrname[10], ch;
        int i;
        clrscr();

        printf("Enter User name: ");
        gets(usrname);
        printf("Enter the password <any 8 characters>: ");

        for(i=0;i<8;i++)
        {
                ch = getch();
                pasword[i] = ch;
                ch = '*' ;
                printf("%c",ch);
        }
        pasword[i] = '\0';
        printf("\n\n\nYour password is :");

        for(i=0;i<8;i++)
        {
                printf("%c",pasword[i]);
        }
}
```

52.   Program to find the length of a string without using the built-in function.

```
# include <stdio.h>

main()
{
        char string[50];
        int  i, length = 0;

        printf("Enter a string\n");
        gets(string);

        for (i=0; string[i] != '\0'; i++) /*keep going through each */
        {                               /*character of the string */
          length++;                     /*till its end */
        }
        printf("The length of a string is the number of characters in it\n");
        printf("So, the length of %s =%d\n", string, length);
}
```

Output:

```
Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5

RUN2
Enter a string
E-Commerce is hot now
The length of a string is the number of characters in it
So, the length of E-Commerce is hot now =21
```

53.   Program to read a matrix A (MxN) and to find the following using functions:
       a) Sum of the elements of each row
       b) Sum of the elements of each column
       c) Find the sum of all the elements of the matrix

       Output the computed results with suitable headings

```
#include <stdio.h>

main()
{
        int arr[10][10];
        int i, j, row, col, rowsum, colsum,sumall=0;
        clrscr();

        printf("Enter the order of the matrix\n");
        scanf("%d %d", &row, &col);

        printf("Enter the elements of the matrix\n");
        for(i=0; i<row; i++)
        {
                for(j=0; j< col; j++)
```

```c
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("Input matrix is\n");
        for(i=0; i<row; i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("%3d", arr[i][j]);
                }
                printf("\n");
        }
                                /* computing row sum */
        for(i=0; i<row; i++)
        {
                rowsum = Addrow(arr,i,col);
                printf("Sum of row %d = %d\n", i+1, rowsum);
        }

        for(j=0; j<col; j++)
        {
                colsum = Addcol(arr,j,row);
                printf("Sum of column  %d = %d\n", j+1, colsum);
        }
                                /* computation of all elements */
        for(j=0; j< row; j++)
        {
                sumall = sumall + Addrow(arr,j,col);
        }
        printf("Sum of all elements of matrix = %d\n", sumall);
}

/* Function to add each row */

int Addrow(int A[10][10], int k, int c)
{
        int rsum=0, i;
        for(i=0; i< c; i++)
        {
                rsum = rsum + A[k][i];
        }
        return(rsum);
}

/* Function to add each column */

int Addcol(int A[10][10], int k, int r)
{
        int csum=0, j;
        for(j=0; j< r; j++)
        {
                csum = csum + A[j][k];
        }
         return(csum);
}
```

Output:

Enter the order of the matrix
3
3

Enter the elements of the matrix
1 2 3
4 5 6
7 8 9

Input matrix is
  1 2 3
  4 5 6
  7 8 9

Sum of row 1 = 6
Sum of row 2 = 15
Sum of row 3 = 24
Sum of column 1 = 12
Sum of column 2 = 15
Sum of column 3 = 18
Sum of all elements of matrix = 45

54.   Program to check whether a given integer number is positive or negative.

```c
#include <stdio.h>

main()
{
        int number;
        clrscr();

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

        if (number > 0)
                printf ("%d, is a positive number\n", number);
        else
                printf ("%d, is a negative number\n", number);

}
```

Output:

Enter a number
-5
-5, is a negative number

RUN2
Enter a number
89
89, is a positive number

55.   Program to find and output all the roots of a quadratic equation, for non-zero coefficients. In case of errors your program should report suitable error message.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main()
{
        float A, B, C, root1, root2;
        float realp, imagp, disc;
        clrscr();

        printf("Enter the values of A, B and C\n");
        scanf("%f %f %f", &A,&B,&C);

        if( A==0 || B==0 || C==0)
        {
                printf("Error: Roots cannot be determined\n");
                exit(1);
        }
        else
        {
                disc = B*B - 4.0*A*C;
                if(disc < 0)
                {
                        printf("Imaginary Roots\n");
                        realp = -B/(2.0*A) ;
                        imagp = sqrt(abs(disc))/(2.0*A);
                        printf("Root1 = %f  +i %f\n",realp, imagp);
                        printf("Root2 = %f  -i %f\n",realp, imagp);
                }
                else if(disc == 0)
                {
                        printf("Roots are real and equal\n");
                        root1 = -B/(2.0*A);
                        root2 = root1;
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
                else if(disc > 0 )
                {
                        printf("Roots are real and distinct\n");
                        root1 =(-B+sqrt(disc))/(2.0*A);
                        root2 =(-B-sqrt(disc))/(2.0*A);
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
        }

}
```

Output:

RUN 1
Enter the values of A, B and C: 3 2 1
Imaginary Roots

Root1 = -0.333333 +i 0.471405
Root2 = -0.333333 -i 0.471405

RUN 2
Enter the values of A, B and C: 1 2 1
Roots are real and equal
Root1 = -1.000000
Root2 = -1.000000

RUN 3
Enter the values of A, B and C: 3 5 2
Roots are real and distinct
Root1 = -0.666667
Root2 = -1.000000

56.   Program to simulate a simple calculator to perform arithmetic operations like addition, subtraction, multiplication and division only on integers. Error message should be reported if any attempt is made to divide by zero.

```c
#include <stdio.h>

main()
{
        char operator;
        float n1, n2;
        float sum, diff, prod, quot, result;

        clrscr();

        printf ("Simulation of Simple Calculator\n\n");

        printf("Enter two numbers: \n");
        scanf ("%f %f", &n1, &n2);

        fflush (stdin);

        printf("Enter the operator [+,-,*,/]: \n");
        scanf ("%c", &operator);


        switch (operator)
        {
                case '+':
                        result = n1 + n2;
                        break;
                case '-':
                        result = n1 - n2;
                        break;
                case '*':
                        result = n1 * n2;
                        break;
                case '/':
                        result = n1 / n2;
                        break;
                default :
                        printf ("Error in operation\n");
```

```
                break;
        }
        printf ("\n%5.2f %c %5.2f= %5.2f\n", n1,operator, n2, result);

}
```

Output:

Simulation of Simple Calculator

Enter two numbers: 3 5
Enter the operator [+,-,*,/]: +

 3.00 + 5.00 = 8.00

RUN2
Simulation of Simple Calculator

Enter two numbers: 12.75 8.45
Enter the operator [+,-,*,/]: -

12.75 - 8.45 = 4.30

RUN3
Simulation of Simple Calculator

Enter two numbers: 12 12
Enter the operator [+,-,*,/]: *

12.00 * 12.00 = 144.00

RUN4
Simulation of Simple Calculator

Enter two numbers: 5 9
Enter the operator [+,-,*,/]: /

 5.00 / 9.00 = 0.56

57.   Program to find the GCD and LCM of two integers output the results along with
the given integers. Use Euclids' algorithm.

```
#include <stdio.h>

main()
{
        int  num1, num2, gcd, lcm, remainder, numerator, denominator;
        clrscr();

        printf("Enter two numbers: \n");
        scanf("%d %d", &num1,&num2);

        if (num1 > num2)
        {
                numerator = num1;
```

```
                denominator = num2;
        }
        else
        {
                numerator = num2;
                denominator = num1;
        }
        remainder = num1 % num2;
        while(remainder !=0)
        {
                numerator   = denominator;
                denominator = remainder;
                remainder   = numerator % denominator;
        }
        gcd = denominator;
        lcm = num1 * num2 / gcd;
        printf("GCD of %d and %d = %d \n", num1,num2,gcd);
        printf("LCM of %d and %d = %d \n", num1,num2,lcm);
}
```

Output:

```
RUN 1
Enter two numbers: 5 15
GCD of 5 and 15 = 5
LCM of 5 and 15 = 15
```

58.   Program to find the sum of odd numbers and sum of even numbers from 1 to N. Output the computed sums on two different lines with suitable headings.

```
#include <stdio.h>

main()
{
        int i, N, oddSum = 0, evenSum = 0;
        clrscr();

        printf("Enter the value of N: \n");
        scanf ("%d", &N);

        for (i=1; i <=N; i++)
        {
                if (i % 2 == 0)
                evenSum = evenSum + i;
        }
        else
                oddSum = oddSum + i;
        printf ("Sum of all odd numbers  = %d\n", oddSum);
        printf ("Sum of all even numbers = %d\n", evenSum);
}
```

Output:

```
RUN1
Enter the value of N: 10
Sum of all odd numbers  = 25
```

Sum of all even numbers = 30

RUN2
Enter the value of N: 50
Sum of all odd numbers = 625
Sum of all even numbers = 650

59.   Program to check whether a given number is prime or not and output the given number with suitable message.

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

main()
{
        int num, j, flag;
        clrscr();

        printf("Enter a number: \n");
        scanf("%d", &num);
        if ( num <= 1)
        {
                printf("%d is not a prime numbers\n", num);
                exit(1);
        }

        flag = 0;
        for ( j=2; j<= num/2; j++)
        {
                if( ( num % j ) == 0)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 0)
                printf("%d is a prime number\n",num);
        else
                printf("%d is not a prime number\n", num);
}
```

Output:

RUN 1
Enter a number: 34
34 is not a prime number

RUN 2
Enter a number: 29
29 is a prime number

60.   Program to generate and print prime numbers in a given range. Also print the number of prime numbers.

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

main()
{
        int M, N, i, j, flag, temp, count = 0;

        clrscr();

        printf("Enter the value of M and N: \n");
        scanf("%d %d", &M,&N);

        if(N < 2)
        {
                printf("There are no primes upto %d\n", N);
                exit(0);
        }
        printf("Prime numbers are\n");
        temp = M;

        if ( M % 2 == 0)
        {
                M++;
        }
        for (i=M; i<=N; i=i+2)
        {
                flag = 0;

                for (j=2; j<=i/2; j++)
                {
                        if( (i%j) == 0)
                        {
                                flag = 1;
                                break;
                        }
                }
                if(flag == 0)
                {
                        printf("%d\n",i);
                        count++;
                }
        }
        printf("Number of primes between %d  and %d = %d\n",temp,N,count);
}
```

Output:

Enter the value of M and N: 15 45

Prime numbers are
17
19
23

29
31
37
41
43
Number of primes between 15 and 45 = 8

61.   Program to read N integers (zero, +ve and -ve) into an array A and to

       a) Find the sum of negative numbers
       b) Find the sum of positive numbers and
       c) Find the average of all input numbers

       Output the various results computed with proper headings.

```c
#include <stdio.h>
#define MAXSIZE 10

main()
{
        int array[MAXSIZE];
        int i, N, negsum=0, posum=0, count1=0, count2=0;
        float total=0.0, averg;

        clrscr();
        printf ("Enter the value of N\n");
        scanf("%d", &N);

        printf("Enter %d numbers (-ve, +ve and zero)\n", N);
        for(i=0; i< N ; i++)
        {
                scanf("%d",&array[i]);
                fflush(stdin);
        }
        printf("Input array elements\n");
        for(i=0; i< N ; i++)
        {
                printf("%+3d\n",array[i]);
        }

                        /* Summing  begins */
        for(i=0; i< N ; i++)
        {

                if(array[i] < 0)
                {
                        negsum = negsum +  array[i];
                }
                else if(array[i] > 0)
                {
                        posum = posum + array[i];
                }
                else if( array[i] == 0)
                {
                        ;
                }
```

```
                total = total + array[i] ;
        }

        averg = total / N;
        printf("\nSum of all negative numbers    = %d\n",negsum);
        printf("Sum of all positive numbers    = %d\n", posum);
        printf("\nAverage of all input numbers  = %.2f\n", averg);

}
```

Output:

Enter the value of N: 5
Enter 5 numbers (-ve, +ve and zero)
5
-3
0
-7
6

Input array elements
 +5
 -3
 +0
 -7
 +6

Sum of all negative numbers   = -10
Sum of all positive numbers   = 11

Average of all input numbers   = 0.20

62.   Program to sort N numbers in ascending order using Bubble sort and print both the given and the sorted array with suitable headings.

```c
#include <stdio.h>
#define MAXSIZE 10

main()
{
        int array[MAXSIZE];
        int i, j, N, temp;

        clrscr();

        printf("Enter the value of N: \n");
        scanf("%d",&N);

        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array is\n");
        for(i=0; i<N ; i++)
        {
```

```c
                printf("%d\n",array[i]);
        }
                        /* Bubble sorting begins */
        for(i=0; i< N ; i++)
        {
                for(j=0; j< (N-i-1) ; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                temp       = array[j];
                                array[j]   = array[j+1];
                                array[j+1] = temp;
                        }
                }
        }
        printf("Sorted array is...\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
}
```

Output:

Enter the value of N: 5
Enter the elements one by one
390
234
111
876
345

Input array is
390
234
111
876
345

Sorted array is...
111
234
345
390
876

63.   Program to accept N numbers sorted in ascending order and to search for a given number using binary search. Report success or failure in the form of suitable messages.

```c
#include <stdio.h>

main()
{
        int array[10];
        int i, j, N, temp, keynum, ascending = 0;
        int low,mid,high;
```

```c
clrscr();

printf("Enter the value of N: \n");
scanf("%d",&N);

printf("Enter the elements one by one\n");
for(i=0; i<N ; i++)
{
        scanf("%d",&array[i]);
}
printf("Input array elements\n");
for(i=0; i<N ; i++)
{
        printf("%d\n",array[i]);
}
                        /* Bubble sorting begins */
for(i=0; i< N ; i++)
{
        for(j=0; j< (N-i-1) ; j++)
        {
                if(array[j] > array[j+1])
                {
                        temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
                }
        }
}
printf("Sorted array is...\n");
for(i=0; i<N ; i++)
{
        printf("%d\n",array[i]);
}

printf("Enter the element to be searched\n");
scanf("%d", &keynum);

                /* Binary searching begins */
low=1;
high=N;
do
{
        mid= (low + high) / 2;
        if ( keynum < array[mid] )
                high = mid - 1;
        else if ( keynum > array[mid])
                low = mid + 1;
} while( keynum!=array[mid] && low <= high);          /* End of do- while */

if( keynum == array[mid] )
{
        printf("SUCCESSFUL SEARCH\n");
}
else
        printf("Search is FAILED\n");
}
```

Output:

Enter the value of N: 4

Enter the elements one by one
3
1
4
2

Input array elements
3
1
4
2

Sorted array is...
1
2
3
4

Enter the element to be searched
4

SUCCESSFUL SEARCH

64.   Program read a sentence and count the number of number of vowels and consonants in the given sentence. Output the results on two lines with suitable headings.

```c
#include <stdio.h>

main()
{
        char sentence[80];
        int i, vowels=0, consonants=0, special = 0;
        clrscr();
        printf("Enter a sentence\n");
        gets(sentence);

        for(i=0; sentence[i] != '\0'; i++)
        {
                if((sentence[i] == 'a'||sentence[i] == 'e'||sentence[i] == 'i'||
                sentence[i] == 'o'||sentence[i] == 'u') ||(sentence[i] == 'A'||
                sentence[i] == 'E'||sentence[i] == 'I'|| sentence[i] == 'O'||
                sentence[i] == 'U'))
                {
                        vowels = vowels + 1;
                }
                else
                {
                        consonants = consonants + 1;
                }
                 if (sentence[i] =='\t' ||sentence[i] =='\0' || sentence[i] ==' ')
                {
                        special = special + 1;
```

```
            }
        }
        consonants = consonants - special;
        printf("No. of vowels in %s = %d\n", sentence, vowels);
        printf("No. of consonants in %s = %d\n", sentence, consonants);
}
```

Output:

Enter a sentence

Good Morning

No. of vowels in Good Morning = 4

No. of consonants in Good Morning = 7

IMPORTANT REVIEW QUESTIONS

1. Distinguish between getchar() and gets() functions.

getchar()	gets()
Used to receive a single character.	Used to receive a single string, white spaces and blanks.
Does not require any argument.	It requires a single argument.

2. Distinguish between scanf() and gets() functions.

scanf()	gets()
Strings with spaces cannot be accessed until ENTER key is pressed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character data type can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers.	Only one string can be received at a time. Strings, floats can be received at a time.

3. Distinguish between printf() and puts() functions.

puts()	printf()
They can display only one string at a time.	They can display any number of characters, integers or strings a time.
All data types of considered as characters.	Each data type is considered separately depending upon the conversion specifications.

4. What is the difference between a pre increment and a post increment operation?

A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1.

5. Distinguish between break and continue statement.

Break	Continue
Used to terminate the loops or to exist loop from a switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop containing it.	The continue statement when executed cause immediate termination of the current iteration of the loop.

6. Distinguish between while and do-while loops.

While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

7. Distinguish between local and global variables

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

8. State the differences between the function prototype the function definition

Function prototype	Function Definition
It declares the function.	It defines the function.
It ends with a semicolon.	It doesn't ends with a semicolon.
The declaration needn't include parameters.	It should include names for the parameters.

9. Compare and contrast recursion and iteration

Both involve repetition.
Both involve a termination test.
Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

10. State the uses of pointers.

- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

11. What is the difference between the address stored in a pointer and a value at the address?

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address, which itself is stored in a pointer.

12. What is the difference between the indirection operator and the address of operator?

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

13. State the difference between call by value and call by reference.

Call by value	Call by reference
int a;	int a;
Formal parameter 'a' is a local variable.	Formal parameter is 'a' local reference.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

14. State the advantages of using bit-fields

- To store Boolean variables (true/false) in one byte
- To encrypt certain routines to access the bits within a byte
- To transmit status information of devices encoded into one or more bits within a byte.

15. State the difference between arrays and structures.

Arrays	Structures
A array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.
Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we can dealing with structures.

Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are not stored in sequence of memory locations.
The array elements are accessed by its followed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.

Its general format is data type variable name [size];	Its general format is:
	struct <struct name>
	{
	    data_type structure member 1;
	    data_type structure member 2;
	      .
	      .
	      .
	    data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	    char studname [25];
	    int rollno;
	} stud1;

16. State the difference between structures and unions

Structures	Unions
Each member in a structure occupies and uses its own memory space	All the members of a union use the same memory space
The keyword struct tells us that we are dealing with structures	The keyword union tells us that we are dealing with unions
All the members of a structure can be initialized	Only the first member of an union can be initialized
More memory space is required since each member is stored in a separate memory locations	Less memory space is required since all members are stored int eh same memory locations

Its general format is:
struct <struct Name>
{
   data_type structure Member1;
   data_type structure Member2;
     .
     .
   data_type structure Member N;
} structure variable;
	Its general formula is:
	union <union Name>
	{
	   data_type structure Member1;
	   data_type structure Member2;
	     .
	     .
	   data_type structure Member N;
	} union variable;

Example:
struct student
{
    char studname[25];
    int rollno;
} stud1;
	Example:
	union student
	{
	    char studname[25];
	    int rollno;
	} stud1;

17. Advantages of functions:

    1. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

    2. The length of source program can be reduced by using function by using it at different places in the program according to the user's requirement.

    3. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

    4. A function can be used by many programs.

    5. Function increases the execution speed of the program and makes the programming simple.

    6. By using the function, portability of the program is very easy.

    7. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

    8. Debugging (removing error) becomes very easier and fast using the function sub-programming.

    9. Functions are more flexible than library functions.

    10. Testing (verification and validation) is very easy by using functions.

    11. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

18. A complete summarized table to represent the lifetime and visibility (scope) of all the storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local