PROGRAMMING LANGUAGES (C-programming)

Unit One

Problem Solving Using Computer

Before writing a computer program, we must be clear about the processing steps to be performed by the computer. Hence, in order to ensure that the program/instructions are appropriate for the problem and are in the correct sequence, program must be planned before they are written. The problem solving contains a number of steps which are as below.

Steps in problem solving by Computer

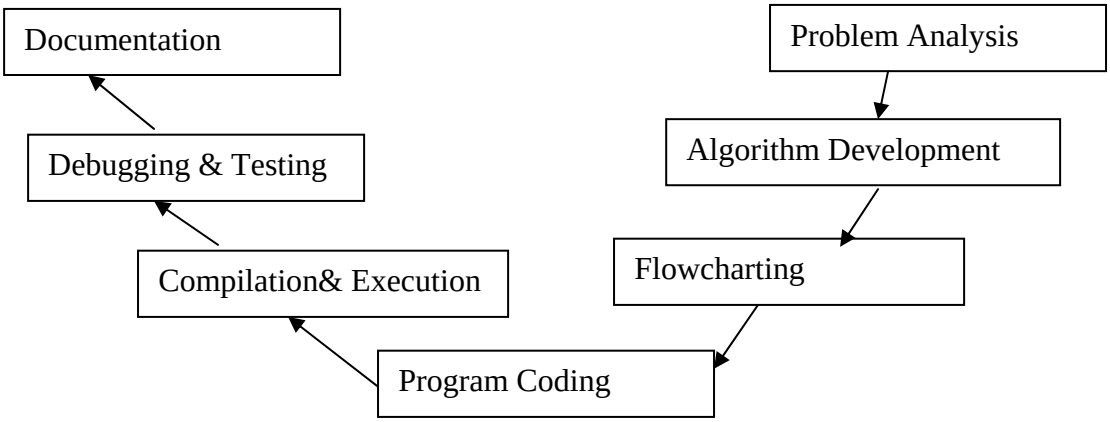


Fig: Steps in problem solving by a computer

1) Problem Analysis:

Before solving a problem, it should be analyzed well. It is impossible to solve a problem by using computer without clear understanding & identification of the problem. Thus, problem analysis contains different activities like determining input/output, software and hardware requirement, time constraints, different types of users, accuracy etc.

2) Algorithm Development:

Algorithm is step by step description of the method to solve a problem. In other words, it can be defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired result is obtained.

Example: write an algorithm to solve the problem which tests a number for even or odd.

Step1: Start

Step2: Read a number.

Step3: Divide the number by 2 and check the remainder.

Step4: If the remainder in step 3 is zero,

Display the number as even otherwise as odd.

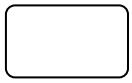
Step5: Stop.

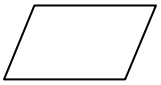
3) Flowchart:

Flowchart is the graphical representation of the algorithm using standard symbols. In other words, flowchart is a pictorial representation an algorithm that uses boxes of different shapes to denote different types of instruction. The actual instructions are written within these boxes using clear statement. The boxes are connected by solid lines having arrow marks to indicate the flow of operation.

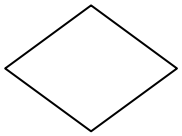
Flowchart

Symbols

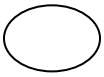
1) Start or End 

2) Input or Output 

3) Processing 

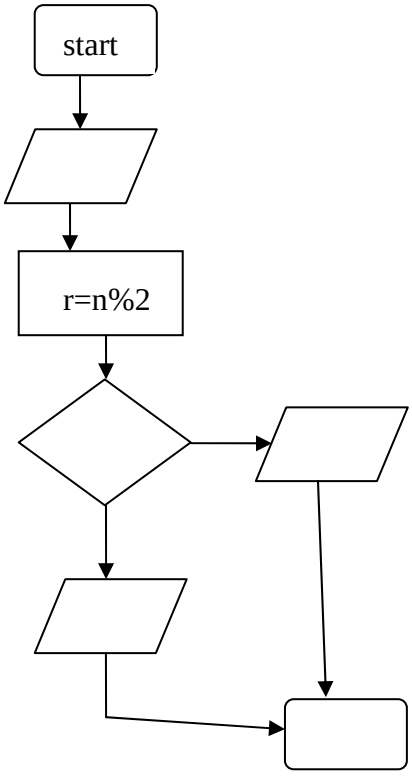
4) Decision 

5) Flow lines

6) ontinuation 

Example of flow chart:

Draw flowchart to solve the problem which tests the number for even or odd.



4) Coding:

This is the process of transforming the program logic design into a computer language format. This stage translates the program design into computer instructions. These instructions are the actual program or software product. It can be said that coding is the act of transforming operations in each box of the flowchart in terms of the statement of the programming.

5) Compilation and Execution:

The process of changing high level language into machine level language is known as compilation. It is done by a compiler. Once the compilation is completed then the program is linked, loaded and finally executed. The original high level language is called the source program and the resulting machine language program is called object program.

6) Debugging & Testing:

Debugging is the discovery and correction of programming errors. Even after taking full care in program design, some errors may remain in the program because the designer might have never thought about a particular case. These errors are detected only when we start executing the program in a computer. Such types of program errors are called BUGS and process of removing these BUGS is knows as debugging.

Testing is the validation of the program. Testing ensures that program performs correctly the required tasks. The program testing and debugging are closely related.

7) Program Documentation:

Documentation of program helps to those who use, maintain and extend the program in future. Properly documented program is useful and efficient in debugging, testing, maintenance and redesign process. A properly documented program can easily be used again when needed and an undocumented program usually requires much extra work. Among the techniques commonly found in documentation are flowcharts, comments, memory maps, and parameter & definition list.

UnitTwo
Introduction to C

Introduction to C:

C is a general-purpose, structured programming language. Its instructions consists ofterms that resemble algebraic expression, augmented by certain English keywords such as if,else, for, do and while, etc. C contains additional features that allow it to be used at a lowerlevel, thus bridging the gap between machine language and the more conventional high levellanguage. This flexibility allows C to be used for system programming (e.g. for writingoperating systems as well as for applications programming such as for writing a program tosolve mathematical equation or for writing a program to bill customers). It also resemblesother high level structure programming language such as Pascal and FORTRAN.

2.1 Historical Development of C:

C was an offspring of the ‘Basic Combined Programming Language’ (BCPL) called B,developed in 1960s at CambridgeUniversity. B language was modified by Dennis Ritchieand was implemented at Bell Laboratories in 1972. The new language was named C. Since itwas developed along with the UNIX operating system, it is strongly associated with UNIX.This operating system was developed at Bell Laboratories and was coded almost entirely inC.

C was used mainly in academic environments for many years, but eventually with therelease of C compiler for commercial use and the increasing popularity of UNIX, it began togain populuidespread support among compiler professionals. Today, C is running under anumber of operating systems including Ms-DOA. C was now standardized by AmericanNational Standard Institute. Such type of C was named ANSI C.

2.2 Importance of C:

Now-a-days, the popularity of C is increasing probably due to its many desirablequalities. It is a robust language whose rich set of built-in functions and operators can be usedof built-in functions and operators can be used to write any complex program. The Ccompiler combines the capabilities of an assemble language with the features of a high-levellanguage and therefore it well suited for writing both system software and business packages.In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types andpowerful operators. It is many times faster than BASIC (Beginners All Purpose SymbolicInstruction Code – a high level programming language).There are only 32 keywords and its strength lies in its built-in functions. Severalstandard functions are available which can be used for developing programs. C is highlyportable. This means that C programs written for one computer can be seen on another withlittle or no modification. Portability is important if we plan to use a new computer with adifferent operating system.C Language is wll suited for structure programming thus requiring the user to think ofa problem in terms of function modules or blocks. A proper collection of these moduleswould make a complete program. This modular structure makes program debugging, testingand maintenance.

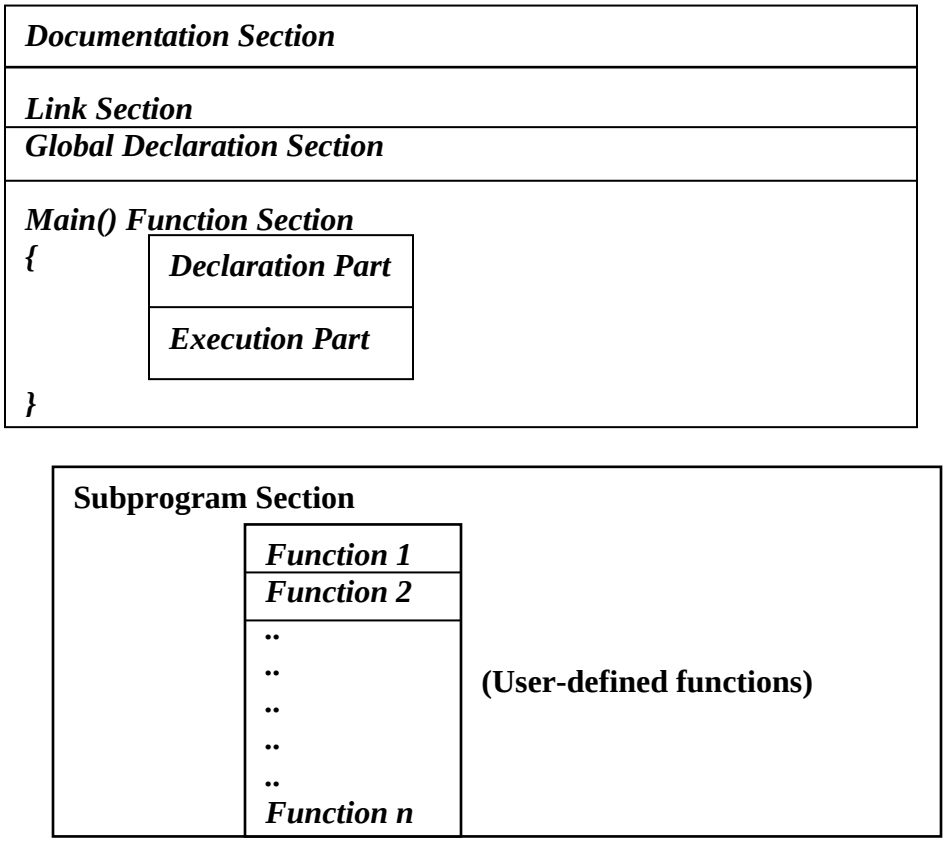
Another important feature of C is its ability to extend itself. A C program is basically acollection of functions that are supported by the C library. We can continuously add our ownfunction to the

C library. With the availability of a large number of functions, theprogramming task becomes simple.

2.3 Basic Structure of C programs:

Every C program consists one or more modules called function. One of the functionmust be called main( ).A function is a sub-routine that may include one or more statementsdesigned to perform a specific task. A C program may contain one or more sections shown in

Fig:



(Fig: Basic Structure of a C program)

The documentation section consists of a set of comment lines giving the name of theprogram, the author and other details which the programmer would like to use later. The linksection provides instructions to the compiler to link function from the system library. Thedefinition defines all the symbolic constants. There are some variables that are used in morethan one function. Such variables are called global variables and are declared in globaldeclaration section that is outside of all the function.Every C program must have one main( ) function section. This section consists

twoparts: declaration part and executable part. The declaration part declares all the variables usedin the executable part. These two parts must appear between the opening and the closingbraces. The program execution begins at the opening braces and ends at the closing brace.The closing brace of the main ( ) function section is the logical end of the program. All thestatements in the declaration and executable parts ends with a semicolon.The subprogram section contains all the user-defined functions that are called in themain ( ) function. User-defined functions are generally placed immediately after the main ( )function, although they may appear in any order. All section, except the main ( ) functionsection may be absent when they are not required.

2.4 Executing a C Program

Executing a program written in C involves a series of steps:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.

The program can be created using any word processing software in non-documentmode. The file name should end with the characters ".c" like program .c, lab1.c, etc. Then thecommand under Ms DOS operating system would load the program stored in the file program.c i.e.

(User-defined Functions)

MSC pay .C

And generate the object code. This code is stored in another file under name *'program.obj'*. In case any language errors are found, the compilation is not completed. The program shouldthen be corrected and compiled again.The linking is done by the commandLINK *program.obj*which generates the executable code with the filename program.exe. Now the command.programwould execute the program and give the results.

/* First Program written in C */

/* Save it as hello.c */

```
# include <stdio.h> /* header file */
Void main ( ) /* main ( ) function */
```

```
{
Print (“hello, world in”), /* statement */
}
Output: hello, world
```

/* Program to calculate the area of a circle */

```
/* area.c */
# include <stdio.h> /* library file access */
Void main( ) /* function heading */
{
float radius, area; /* variable decleration */
printf (“Enter radious?=”) /* output statement */
scanf (“%f”, & radius); /* input statement */
area = 3.14159 * radius; /* assignment statement */
printf (“Area=%f”, area); /* output statement */
}
```

Unit Three

C Fundamentals

C Fundaments is concerned with the basic elements used to construct simple C statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration expressions and statements. The purpose of this material is to introduce certain basic concept and to provide some necessary definitions. A programming language is designed to help process certain kinds of data consisting of numbers, characters and strings and to provide useful output known as information. The task of programming of data is accomplished by executing a sequence of precise instruction called a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax rules.

3.1 Character Set:

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form basic program elements (e.g. constants, variables, operators, expressions, etc). The special characters are listed below: + - * / = % & # ! ? ^ “ ' ~ \ | <> ( ) [ ] { } : ; . , - (blank space) (Horizontal tab) (White Space) Most versions of the language also allow certain other characters, such as @ and $ to be included with strings & comments.

3.2 Identifiers & Keywords:

C Tokens:

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are also known as C tokens. C has six types of tokens:

1. Identifiers e.g. x area __ temperature PI
2. Keywords e.g. int float for while
3. Constants e.g. -15.5 100
4. Strings e.g. “ABC” “year”
5. Operators e.g. + - *
6. Special Symbols e.g. ( ) [ ] { }

Identifiers:

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consisted of letters and digits, in any order, except that first character must be a letter. Both upper and lower case letters are permitted, though common usage favors the use of lowercase letters for most type of identifiers. Upper and lowercase letters are not interchangeable (i.e. an uppercase letter is not equivalent to the corresponding lowercase letters). The underscore (_) can also be included, and considered to be a letter. An underscore is often used in middle of an identifier. An identifier may also begin with an underscore.

Rules for Identifier:

1. First character must be an alphabet (or Underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.

5. Must not contain white space.

The following names are valid identifiers

x a12 sum_1 _temp namearea tax_rate TABLE

The following names are not valid identifier for the reason stated

4th	The first character must be letter
"x"	Illegal characters (")
Order-no	Illegal character (-)
Error-flag	Illegal character (blank space)

Keywords:

There are certain reserved words, called keywords that have standard, predefinedmeanings in C. These keywords can be used only for their intended purpose, they cannot beused as programmer-defined identifiers. The standard keywords are:***auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struc, switch, typedef, union, unsigned, void, volatile, while.***

The keywords are all lowercase. Since upper and lowercase characters are notequivalent, it is possible to utilize an uppercase keyword as an identifier. Normally, however,this is not done, as it is considered a poor programming practice.

3.3 Data Types:

C language is rich in its data types. C supports several different types of data, each ofwhich may be represented differently within the computer memory. There are three cases ofdata types:

1. Basic data types (Primary or Fundamental) e.g.int, char
2. Derived data types e.g. array, pointer, function
3. User defined data types e.g. structure, union, enum

The basic data types are also k/a built in data types. The basic data types are listedbelow. Typical memory requirements are also given:

Data Types Description Typical Memory Requirement

char	single character	1 byte
int	integer quantity	2 bytes
float	floating-point number	4 bytes (1 word)
double	double-precision floating point number	8 bytes (2 word)

In order to provide some control over the range of numbers and storage space, C hasfollowing classes: signed, unsigned, short, and long.

Types	**Size**
char or signed char	1 byte
unsigned char	1 byte
int	2 bytes
short int	1 byte
unsigned short int	1 byte
signed int	2 bytes
unsigned int	2 bytes
long int	4 bytes
signed long int	4 bytes
unsigned long int	4 bytes
float	4 bytes
double	8 bytes
long double	10 bytes

void is also a built-in data type used to specify the type of function. The void type hasno values.

3.4 Constants, Variables

Constants in C refer to fixed values that do not change during the execution of aprogram. There are four basic types of constants in C. They are integer constants, floatingpoint constants, character constants and string constants.Integer and floating point constants represent numbers. They are often referred tocollectively as numeric _ type constants. The following rules apply to all numeric typeconstants.

1. Commas and blank spaces cannot be included within the constants.
2. The constant can be preceded by a minus (-) if desired. The minus sign is anoperator that changes the sign of a positive constant though it can be thought of as apart of the constant itself.

3. The value of a constant cannot exceed specified minimum and maximum bounds.

For each type of constant, these bounds will vary from one C compiler to another.

Integer Constants:

An integer constant is an integer-valued number. Thus it consists of a sequence ofdigits. Integer (number) constants can be written in three different number systems: decimal(base 10), octal (base 8) and hexadecimal (base 16). Beginning programmers rarely howeveruse anything other than decimal integer constants.A decimal integer constant can consists of any combination of digits taken from the set0 through 9. If the constant contains two or more digits, the first digit must be somethingother than 0. Several valid decimal integer constants are shown below:

0 1 143 5280 12345 9999

The following decimal integer constants are written incorrectly for reason stated:

12,452	illegal character (,)
36.0	Illegal character (.)
10 20 30	illegal character (blankspace)
123_45_6743	illegal character (-)
900	the first digit cannot be zero.

An octal integer constant can consist of any combination of digits taken from the set 0through 7. However, the first digit must be 0, in order to identify the constant as an octalnumber.

Valid octal number (integer) constants are shown below:

0 01 0743 07777

The following octal integer constants are written incorrectly for the reason stated:

743	does not begin with 0.
05280	illegal character (8)
777.777	illegal character (.)

A hexadecimal integer constant must begin with either 0_x or 0X. It can then be followedby any combination of digits taken from the sets 0 through 9 and a through f (either upper orlower case). The letters a through f (or A through F) represent the (decimal) quantities 10through 15 respectively. Several valid hexadecimal integer constants are shown below:

0_x 0X1 0X7FFF 0xabcd

The following hexadecimal integer constants are written incorrectly for the reasonstated:

0X12.34 illegal character (.)

013E38 doesn't begin with 0_x or 0X.

0_x.4bff illegal character (.)

0XDEFG illegal character(G)

Unsigned and Long Integer Constants:

Unsigned integer constants may exceed the magnitude of ordinary integer constants byapproximately a factor of l, though they may not be negative. An unsigned integer constantcan be identified by appending the letter ( ) (either upper or lowercase) to the end of theconstant.
Long integer constants may exceed the magnitude of ordinary integer constants, butrequire more memory within the computer. A long integer constant can be identified byappending the letter L (either upper or lowercase) to the end of the constant.An unsigned long integer may be specified by appending the letters UL to the end of theconstant. The letters may be written in either upper or lowercase. However, the U must precede the L.

Several unsigned and long integer constants are shown below:

Constant	Number System
50000 U	decimal (unsigned)
123456789 L	decimal (long)
123456789 UL	decimal (unsigned long)
0123456 L	octal (long)
0777777 U	octal (unsigned)
0X50000 U	hexadecimal (unsigned)
0XFFFFFUL	hexadecimal (unsigned long)

Floating Point Constants:

A floating point constant is a base 10 number that contains either a decimal point or anexponent or both).
Several valid floating point constants

0.	1.	0.2	827.602
500.	0.000743	12.3	
2E.8	0.006e.3	1.6667e+8	

The following are not valid floating point constants for the reason stated.

1	Either a decimal point or an exponent must be present.
1,000.0	Illegal character (,)
2E+10.2	The exponent must be an integer (it cannot contain a decimalpoint)
3E 10	Illegal character (blank space) in the exponent.

The quantity 3×10^5 can be represented in C by any of the following floating pointconstants:

300000.	3e5	3e+5	3E5	3.0e+5
.3e5	0.3E6	30E4	30.E4	300e3

Character Constants:

A character constant is a single character, enclosed in apostrophes (i.e. single quotationmarks).

Several character constants are shown below:

‘A’ ‘X’ ‘3’ ‘?’‘ ´

Character constants have integer values that are determined by the computer’s particularcharacter set. Thus, the value of a character constant may vary from one computer to another.The constants themselves, however, are independent of the character set. This featureeliminates the dependence of a C program on a particular character set.Most computers, and virtually all personal computers make use of ASCII (i.e. AmericanStandard Code for Information Interchange) character set, in which each individual characteris numerically encoded with its own unique 7-bit combination (hence a total of 2_7=128difference characters).

Several character constant and their corresponding values, as defined by ASCIIcharacter set areshown below:

Constant	**Value**
‘A’	65
‘X’	120
‘3’	51
‘?’	63

‘ ’ 32

These values will be the same for all computers that utilize the ASCII character set.

String Constants:

A string consists of any number of consecutive characters (including none), enclosed in(double) quotation marks. Several string constants are shown below:

“green”	“Washington, D.C. 2005”	“207-32-345”
“$19.95”	“THE CORRECT ANSWER IS”	“2*(I+3”
“ ”	“Line 1\n Line 2\n line 3”	“ ”

The string constants “Line 1\n Line 2\n Line 3” extends over three lines, because of thenewlinecharacters that are embedded within the string. Thus, the string would be displayedas

Line 1

Line 2

Line 3

The compiler automatically places a null character (\0) at the end of every stringconstant, as the last character within the string (before the closing double quotation mark).This character is notvisible when the string is displayed.A character constant (e.g. ‘A’) and the corresponding single-character string constant(“A’) are not equivalent. A character constant has an equivalent integer value, whereas asingle character string constant does not have an equivalent integer value and in fact, consistsof two characters – the specified character followed by the null character (\o).

Variables:

A variable is an identifier that is used to represent some specified type of informationwithin a designated portion of the program. In its simplest form, a variable is an identifier thatis used to represent a single data item, i.e., a numerical quantity or a character constant. Thedata item must be assigned to the variable at some point in the program. A given variable canbe assigned different data items at various places within the program. Thus, the informationrepresented by the variable can change during the execution of the program. However, thedata type associatedwith the variable are not change.A C program contains the following lines:

```
int a,b,c ;
char d ;
```

```
- - - - -
- - - - -
a = 3 ;
b = 5 ;
c = a + b ;
d = ‘a’ ;
- - - - -
- - - - -
a = 4 ;
b = 2 ;
c = a – b ;
d= ‘w’
```

The first two lines are not type declaration which state that a, b and c are integervariables, and that d is a character type. Thus, a, b and c will each represent an integer-valuedquantity, and d will represent a single character. The type declaration will apply throughoutthe program.The next four lines cause the following things to happen: the integer quantity 3 isassigned to a, 5 isassigned to b and the quantity represented by the sum a+b (.e. 8) isassigned to c. The character ‘a’ is assigned then assigned to d.In the third line within this group, the values of the variables aand b are accessedsimply by writing the variables on the right-hand side of the equal sign.The last four lines redefine the values assigned to the variables as the integer quantity 4is assigned to a, replacing the earlier value, 3; then 2 is assigned to b, replacing the earliervalue, 5; The difference between a and b (i.e. 2) is assigned to c, replacing the earlier value 8.Finally the character ‘w’ is assigned to d, replacing the earlier character, ‘a’.

3.5 Declarations

A declaration associates a group of variables with a specific data type. All variablesmust be declared before they can appear in executable statements.A declaration consists of a data type, followed by one or more variable names, endingwith a semicolon. Each array variable must be followed by a pair of square brackets,containing a positive integer which specifies the size (i.e.the number of elements) of thearray.

A C program contains the following type declarations:

int a, b, c ;

float root1, root2 ;

char flag, text [80],

Thus, a, b and c are declared to be integer variables, root1 and root 2 are floatingvariables, flag is a char-type variable and text is an 80-element, char-type array. Squarebrackets enclosing the size specification for text.

These declarations could also have been written as follows:

int a ;

int b ;

int c ;

float root1 ;

float root2 ;

char flag ;

char text [80] ;

 # A C program contains the following type declarations:

short int a, b, c ;

long int r, s, t ;

int p, q ;

Also written as

short a, b, c ;

long r, s, t ;

int p, q ;

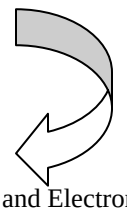
short and short int are equivalent, as are long and long int.

 # A C program contains the following type declarations

float c1, c2, c3 ;

double root1, root2 ;

also written as



long float root1, root2 ;

 # A C program contains the following type declarations.

int c = 12 ;

char star = '*' ;

float sum = 0. ;

double factor = 0.21023e-6

Thus, c is an integer variable whose initial value is 12, star is a char type variableinitially assigned the character '*', sum is a floating point variable whose initial value is 0. ,and factor is double precision variable whose initial value is $0.21023 \times 10_6$.

 # A C program contains the following type declarations.

char text [ ] = "California" ;

This declaration will cause text to be an 11-element character array. The first 10elements will represent the 10 characters within the word California, and the 11_{th} element willrepresentthe null character (\0) which automatically added at the end of the string.

Te declaration could also have been written

char text [11] = "California" ;

Where size of the array is explicitly specified. In such situations, it is important,however, that the size be specified correctly. If the size is too small, eg. ,

char text [10] = "California" ;

The character at the end of the string (in this case, the null character) will be lost. If thesize is too large e.g.,

char text [20] = "California" ;

The extra array elements may be assigned zeros, or they may be filled with meaninglesscharacters.

The array is another kind of variable that is used extensively in C. An array is anidentifier that refers to collection of data items that have the same name. The data items mustall be of the same type (e.g. all integers, all characters). The individual data items arerepresented by their corresponding array element (i.e. the first data item is represented by thefirst array element, etc). The individual array elements distinguished from one another by thevalue that is assigned to a subscript.

c	a	l	i	f	o	r	n	i	a	\0

Subscript:01 2 3 4 5 6 7 8 9 10

3.6 Escape Sequence:

Certain nonprinting character, as well as the backslash (\) and apostrophe (`), can beexpressed in terms of escape sequences. An escape sequence always begins with a backslashand is followed by one or more special characters. For example, a linefeed (LF), this is referred to as a newline in C, can be represented as ln. Such escape sequences alwaysrepresent single characters, eventhough they are written in terms of two or more characters.The commonly used escapesequences are listed below:

Character	Escape Sequence	ASCII Value
bell (alest)	\a	007
backspace	\b	008
horizontal tab	\t	009
vertical tab	\v	011
newline (line feed)	\n	010
form feed	\f	012
carriage return	\r	013
quotation mark (”)	\”	034
apostrophe (`)	\`	039
question mark (?)	\?	063

backslash (\)	\\	092
null	\0	000

Several character constants are expressed in terms of escape sequences are

‘\n’ ‘\t’ ‘\b’ ‘\’’ ‘\\’ **‘\”’**

The last three escape sequences represent an apostrophe, backslash and a quotationmark respectively.

Escape Sequence ‘\0’ represents the null character (ASCII 000), which is used toindicate the end of a string. The null character constant ‘\0’ is not equivalent to the characterconstant ‘0’.

The general form ‘\000’ represents an octal digit (0 through 7). The general form of ahexadecimal escape sequence is \xhh, where each h represents a hexadecimal digit (0 through9 and a through f).

3.7 Preprocessors Directives:

The C preprocessor is a collection of special statements, called directives that areexecuted at the beginning of compilation process. Preprocessors directives usually appear atthe beginning of aprogram. A preprocessor directive may appear anywhere within aprogram. Preprocessordirectives follow special syntax rules that are different from thenormal C syntax. They all beginwith the symbol # in column one and do not require asemicolon at the end. We have already used the directives #define and #include to a limitedextent. A set of commonly used preprocessor directives and their functions are listed below:

Directives	Functions
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Species the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile-time condition

#else	Specifies alternatives when #if test fails.

These directives can be divided into three categories:

1. Macro substitution directives
2. File inclusion directives
3. Compiler control directives

Macro substitution directives:

Macro substitution is a process where an identifier in a program is replaced by apredefined string composed of one or more tokens. The general formatis:

#define identifier string
#define PI 3.1415926
#define FALSE 0
#define COUNT 100
#define CAPITAL “KATHMANDU”

File inclusion directives

We use File inclusion in order to include a file where functions or macros are defined.

*#include <filename>*or *#include “filename”*

Compiler control directives:

In order find files based switch or (on or off) particular line or groups of lines in aprogram, we use conditional compilation. For that, we use following preprocessor directivessuch as #ifdef, #endif, #ifndef, #if,#else.

3.8 Typedef Statement:

C supports a feature known as “type definition” that allows users to define an identifierthat would represent an existing data type. The user-defined data type identifier can later beused to declare variables. It takes the general form:

typedef type identifier ;

Where type refers to an existing data type and identifier refers to the “new” name givento the data type. The existing data type may belong to any class of type, including the userdefined ones. The new type is new only in name, but not the data type. typedef cannot createa new type. Some examples of type definition are:

```
typedef int units ;
typedef float marks ;
```

Here, units represent int and marks represents float. They can be later used to declarevariables as follows:
units batch1, batch2 ;
marks name1 [50] ; name2[50] ;
batch1 and batch2 are declared as int variable and name1[50] and namer[50] aredeclared as 50 element floating point array variables. The main advantage of typedef is thatwe can create meaningful data type names for increasing the readability of the program.

3.9 Symbolic Constants:

A symbolic constant is a name that substitutes for a sequence of characters. Thecharacters may represent numeric constant, a character constant or a string constant. Thus, asymbolic constant allows a name to appear in place of a numeric constant, a characterconstant or a string. When a program is compiled, each occurrence of a symbolic constant isreplaced by its corresponding character sequence.Symbolic constants are usually defined at the beginning of a program. The symbolicconstants may then appear later in the program in place of the numeric constants, characterconstant, etc that the symbolic constants represent.
A symbolic constant is defined by writing
#define name text

Where name represents a symbolic name, typically written in uppercase letters, and textrepresents the sequence of characters that is associated with the symbolic name. It doesn’trequire semicolon. For example

#define TAXRATE 0.13

#define PI 3.141593

#define TRUE 1

#define FALSE 0

#define FRIEND “Susan”

area = PI * radius * radius; is equivalent to

area = 3.141593 * radius * radius ;

**********END UNIT #3**********

Unit Four

Operators & Expressions

Individual constants, variables, array elements and function references can be joinedtogether by various operators to form expression. C includes a number of operators which fallinto several different categories. Such an arithmetic operators, unary operators, rational andlogical operators,assignment operators and the conditional operators, bitwise operator.

The data items that operators act upon are called operands. Some operators require twooperands, while other act upon only one operand. Most operators allow the individualoperands to be expressions. A fewoperators permit only single variables as operands.

4.1 Operators

Arithmetic Operators:

There are five arithmetic operators in C. They are

Operators	Purposes
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	remainder after integer division (also called modulus operator)

There is no exponential operator in C. However, there is a library function (pow) tocarry out exponential.

The operands acted upon by arithmetic operators must represent numeric values. Theremainder operator (%) requires that both operands be integers and the second operand benon zero. Similarly, the division operator (/) requires that the second operand be non-zero.Division of one integer quantity by another is referred to as integer division. Theoperation always result in a truncated quotient (i.e. the decimal portion of the quotient will bedropped). On the other hand, if a division operation is carried out with two floating-pointnumbers, or one floating point numbers and other integer, the result will be a floating point.

4.2 Precedence & Associativity:

Precedence of an operator in C is defined as the order of evaluation. The operators aregrouped hierarchically according to their precedence. Operations with a high precedence arecarried out before operation having a lower precedence. The natural order of evaluation canbe altered using parenthesis.

Table : Precedence of Arithmetic Operators

Precedence	Arithmetic Operators	Descriptions
1	* / %	multiplication, division, modular division
2	+ -	addition, subtraction

Associativity:

The order, in which consecutive operations within the same precedence group arecarried out, is known as associativity. Within each of the precedence groups described above,the associativity is left to right. In other word, consecutive addition and subtractionoperations are carried out from left to right, as are consecutive multiplication, division andremainder operations.

a = 10, b = 3

Expression	**Value**
a+b	13
a-b	7
a*b	30
a/b	3
a%b	1

Determine the hierarchy of operations and evaluate the following expressions, assumingthat **i** is an integer variables:

i = 2*3/4+4/4+8-2+5/8

Stepwise evaluation of this expression is shown below:

i = 2*3/4+4/4+8-2+5/8	
i = 6/4+4/4+8-2+5/8	Operation : *
i = 1+4/4+8-2+5/8	Operation : /
i = 1+1+8-2+5/8	Operation : /
i = 1+1+8-2+0	Operation : /
i = 2+8-2+0	Operation : +
i = 10-2+0	Operation : +

i = 8+0 Operation : -

i = 8 Operation : +

Determine the hierarch of operations and evaluate the following expression, assumingthat **k** is a float integer variable.

k = 3/2*4+3/8+3

Solution: Stepwise

k = 3/2*4+3/8+3

= 1*4+3/8+3 Operation : /

= 4+3/8+3 Operation : *

= 4+0+3 Operation : /

= 4+3 Operation : +

= 7 Operation : +

Suppose x is integer & evaluate x=9-12/(3+3)*(2-1)

Step1 : x = 9-12/6*(2-1) operation : (+)

Step2 : x = 9-12/6*1 operation : (-)

Step3 : x = 9-2*1 operation : (/)

Step4 : x = 9-2 operation : (*)

Step5 : x = 7 operation : (-)

Relational and Logical Operators:

Relational Operators are those that are used to compare two similar operands, anddepending on their relation take some actions. The relational operators in C are listed as

Operators	**Meaning**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
= =	Equality equal to
! =	Operators not equal to

These operators all fall within the same precedence group, which is lower than thearithmetic and unary operators. The associativity of these operators is left to right.

Suppose that i, j and k are integer variables whose values are 1, 2 and 3 respectively.Several relational expressions involving these variables:

Expressions	**Interpretation**	**Value**
i < j	true	1
(i+j) >= k	true	1
(j+k) > (i+5)	false	0
k! = 3	false	0
j = = 2	true	2

In addition to the relational and equality operators, is used to assign a value to anidentifier, where as the equality operator is used to determine if two expressions have thesame value. These operators cannot be used in place of one another.

Assignment expressions are often referred as assignment statements.

Multiple assignments of the form are permissible as

identifier1 = identifier2 = - - - - - - - = identifier n

In such situation, the assignments are carried out from right to left.

Multiple assignment

identifier1 = identifier = expression.

is equivalent to

identifier1 = (identifier2 = expression)

And so on, with right to left resting for additional multiple assignments. C contains the following five additional assignment operators:

+=, - = , * = , / = and % =

They are also k/shorthand assignment operators.

expression1 + = expression2

Is equivalent to:

expression1 = expression 1 + expression

Expression Equivalent Expression

a + = b a = a+b

a - = b a = a-b

a * = b a = a*b

a / = b a = a/b

a% = b a = a%b

The general form of shorthand assignment operators

variable1 operator = variable2 (or expression)

The use of shorthand assignment operator has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier towrite.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

Unary Operators:

C includes a class of operators that act upon a single operand to produce a new value.Such operators k/a unary operators. Unary operators usually precede their single operands,though some unary operators are written after their operands.* There are two other type ofunary operator also available in C:

1. Increment Operator (+ +)
2. Decrement Operator (- -)

The increment operator causes its operand to be increased by 1 where as the decrementoperator causes its operand to be decreased by 1. The operand used with each of theseoperators must be a single variable.

C contains three **logical operators**:

Operator	**Meaning**
&&	And
||	Or
!	Not

Logical operators are used to compare & evaluate logical and relational expressions.Operator && is referred as logic and, the operator || is referred as logic or.The result of a logic and operation will be true only if both operands are true where asthe result of a logic or

operation will be true if either operand is true or if both operands aretrue. In other word, the result of a logic or operation will be false only if both operands arefalse.

Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5 and c is a character variable that represents the character ‘w’. Severalcomplex logical expressions that make use of these variables are shown below:

Expression	**Interpretation**	**Value**
(i>=6) && (c= = ‘w’)	true	1
(i>=6) || (c = = 119)	true	1
(f<11) && (i>100)	false	0
(c! = ‘p’) || (iff<=10)	true	1
!(i>(f+1))	false	0

C also includes the unary operator ! that negates the value of a logical expression, i.e. itcauses an expression that is originally true to become false and vice-versa. This operator isreferred to as the logical negative (or logical not) operator.

Assignment Operators:

There are several different assignment operators in C. All of them are used to formassignment expressions which assign the value of an expression to an identifier.The most commonly used assignment operator is =. Assignment expressions that makeuse of the operator are written in the form:

identifier = expression

Where identifier generally represents a variable, and expression represents a constant, avariable or more complex expression.

Here are some typical assignment expressions that make use of the = operator.

```
a = 3
x = y
delta = 0.001
sum = a+b
area = length * width
```

Assignment operator = and equality operator = = are distinctly different. Theassignment operator can be applied only to integer type signed or unsigned) and not to floator double. They are

Operator	**Meaning**
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	Left shift
>>	right shift
~	bitwise one's complement operator

Consider, a = 60 and b = 15. THe binary representation of a and b for 32 bits

a = 0000 0000 0011 1100
b = 0000 0000 0000 1111
c = a & b = 0000 0000 0011 1111 = 12
d = a | b = 0000 0000 0011 1111 = 63
n = ~ a = 1111 1111 1100 0011 = -61
e = a ^ b = 0000 0000 0011 0011 =51

For Bitwise Shift Operator
Operand Bitwise Shift operator number
For e.g.
a = 0000 0000 0011 1100

f = a << 3
shift1 = 0000 0000 0111 1000
shift2 = 0000 0000 1111 0000
shift3 = 0000 0001 1110 0000 = f = 480
Similarly,
g = a >> 3

shift1 = 0000 0000 0001 1110

shift2 = 0000 0000 0000 1111

shift3 = 0000 0000 0000 0111 = g = 7

Special Operators:

C supports some special operators such as comma operator, size of operator, pointeroperator (* and &) and member selection operators. (.and)⟶

Comma Operator:

The comma operator can be used to link the related expression together. Comma linked lists of expression are evaluated left to right and the value of right-most expression isthe value of combined expression. For example,value = (x = 10, y = 5, x + y),

Here, 10 is assigned to x and 5 is assigned to y and so expression x+y is evaluated as(10+5)i.e. 15.

Size of Operator:

The size of operator is used with an operand to return the number of bytes it occupies. Itis a compile time operand. The operand may be a variable, a constant or a data type qualifier.The associativity of size of right to left. For example Suppose that i is an integer variable, x is a floating-point variable, d is double-precisionvariable and c is character type variable.

The statements:

printf (“integer : %d \n”, size of i) ;

printf (“float : %d \n”, size of x) ;

printf (“double : %d \n”, size of d) ;

printf (“character : %d \n”, size of c) ;

Unary expression	**Equivalent expression**
++ variable or (variable ++)	variable = variable +1
-- variable or (variable _ _)	variable = variable -1

The investment or decrement operators can each be utilized two different ways,depending on whether the operator is written before or after. If the operator is written beforethe operand then it is called as prefix unary operator. If the operator is written after theoperand, then it is called as postfix unary operator. When prefix is used, the operand will bealtered in value before it is utilized for its intended purpose within the program. Similarly,when postfix is used, the operand will be altered in value after it is utilized.A C program includes an integer variable i whose initial value is 1. Suppose theprogram includes the following three printf statements.

```
printf (“i = %d\n”, i) ;
printf (“i = %d\n”, ++i) ;
printf (“i = %d\n”, i) ;
```

These printf statements will generate the following three lines of output:

```
i = 1
i = 2
i = 2
```

Again, suppose

```
printf (“i = %d\n”, i) ;
printf (“i = %d\n”, i++) ;
printf (“i = %d\n”, i) ;
```

These statements will generate the following three lines of output:

```
i = 1
i = 1
i = 2
```

The precedence of unary increment or decrement is same and associativity is right toleft.

Conditional Operators:

The operator ?: is known as conditional operator. Simple conditional operations can becarried out with conditional operator. An expression that make use of the conditional operatoris called a conditional expression. Such an expression can be written in place of traditional ifelsestatement.

A conditional expression is written in the form:

expression1 **?** expression2 **:** expression3

When evaluating a conditional expression, expression1 is evaluated first. If expression1is true, the value of expression2 is the value of conditional expression. If expression1 is false,the value of expression3 is the value of conditional expression.

For example:

```
a = 10 ;
b = 15 ;
x = (a > b) ? a : b ;
```

In this example, x will be assigned the value of b. This can be achieved by using theif_else statement as follows:

```
if (a<b)
x = a ;
else
x = b ;
```

Bitwise Operators:

Bitwise Operator are used for manipulating data at bit level. These operators are usedfor testing the bits, or shifting them right or left. Bitwise

The most common used unary operator is unary minus, where a numerical constant, variableexpression is preceded by a minus sign.

Unary minus operator is distinctly different from the arithmetic operator which denotessubtraction (-). The subtraction operator requires two separate operands.

Several examples of unary minus operator

-743 - 0X7FFF -0.2 -5E-8

-r00+1 - (x+y) -3 *(x+y)

In C, all numeric constants and variable are considered as positive. So, unary plus is notwritten as unary minus such as

+743 = 743

+(x+y) = x+y

**********END UNIT #4**********

Unit Five

Input and Output(I/O) Functions

Reading the data from the input devices and displaying the result on the screen, are the two main task of any program. To perform these tasks,C has a number of input and output functions. When a program needs data, it takes the data through the input functions and sends results obtained through the output functions. Thus the input/output functions are the link between the user and the terminal.There are number of input/output functions in c based on the data types. The input/output functions are classified into two types.

Program

1) Formatted functionsInput ⟶ Output ⟶

2) Unformatted functions.

1) Formatted Functions:

The input function scanf() and output function printf() fall under this category. These formatted functions allow us to input or output in a fixed or required format. For example, while displaying value of certain variable, we can specify the number of decimal places after decimal points, number of spaces before the value, the position where the output is to be displayed etc.

printf() function:

Printf() is a built in function which is used to output data from the computer onto a standard output device i.e. monitor's screen. This function can be used to output any combination of numerical values, single character and strings. Generally, the printf() function is written as

Printf(control string, arg1, arg2, arg3,…..,argn);

Where control string refers to a string that contains formatting information and arg1, arg2, arg3,…….,argn are arguments that represent the individual data items. The arguments can be written as constants, single variable or array names or more complex expression. Here, control string consists of individual groups of character, with one group of character for each output data item. Each character group must begin with a percent sign (%). The control string can be written in following form.

Control string= %[flags][field width][.precision] conversion character

Where the items within [] are optional and % and conversion character are compulsory.

Flags: The flags affect the appearance of the output. The flags must be placed immediately after the percent sign. The flags may be -, +, 0

- - : - data item is left justified.
- + : - A sign will precede each signed numerical data item.
- 0 : - Leading 0s are appeared instead of leading blanks.

e.g. let a=45;

printf("%05d");

Output: 00045 // leading 0s

Field width:

It sets the minimum field width. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters in the data item exceeds the specified field width, then additional space will be allocated to the data item so that the entire data item will be displayed.

e.g.

a=45;

printf("%5d",a);

output: - - -45 i.e. minimum width is 5. (here – represents space)

precision:

It sets the number of digits after decimal point.

a=45.674;

printf("%.2f",a);

Output:45.67

Conversion character:

d for integer data

f for float data

c for character

etc……

scanf() function:

This built-in function can be used to enter input data into the computer from a standard input device i.e. keyboard. The function can be used to enter any combination of numerical

values, single characters and strings. In other word, it is used for runtime assignments of variables. In general term, the scanf() function is written as

scanf(control string, arg1, arg2, ….,argn);

Where control string refers to a string containing certain required formatting information and arg1, arg2, ….,argn are arguments that represent the individual input data items. The arguments represent pointer that indicate the address of the data items within computer memory. Thus, it is preceded by ampersand i.e. &.

The control string consists of individual groups of characters, with one group for each input data item. Each character group must begin with percent sign (%).

Control string= %[field width] conversion character.

Filed width:

It limits the number of input characters (i.e. it sets the maximum number of characters to be entered). This is an unsigned integer which is placed within the control string between the % and the conversion character. The data item may contain fewer characters than the specified field width. However, the number of characters in the actual data item can not exceed the specified field width. Any characters that extend beyond the specified field width will not be read. This is optional field in scanf() function.

e.g.
```
int a;
scanf("%3d",&a);
printf("\n%d",a);
```

Output:

23456 // say input data by user

234 //output

Conversion character:

It is same as in the case of printf() function.

An example showing its syntax:

scanf ("%d %f %c", &a, &b, &c);

Where' &' is the address operator, also called ampersand operator used to indicate the memory location of defined variables and a is integer, b is float and c is character type variable.

When user enters inputs as 34 89.9 r, then it assigns variables as

a=34, b=89.9 and c='r'

2) Unformatted Functions:

There are several functions under this category which don't allow us to enter or display the various data items in required format. The functions getchar(), putchar(), gets(), puts(), getch(), getche(), putch() are considered as unformatted functions.

getchar():

It returns a single character from a standard input device. In general term, it is written as

character variable = getchar();

Here, character variable refers to some previously declared character variable**.**

putchar();

This functions prints one character on the screen at a time which is read by the standard input.In general term, it is written as

putchar(character variable);

Here, character variable refers to some previously declared character variable**.**

e.g.

```
char c;
c=getchar();
putchar(c);
```

Output:

```
r_ // input
r  // output
```

getch(), getche() and putch():

Thefunctions getch() and getche() read single character the instant it is typed without waiting for the enter key to be hit. The difference between them is that getch() just reads the character that typed without echoing it on the screen and getche() reads the character and it echoes (displays) the character that typed to the screen.

Again, the function putch() prints a character taken by the standard input device

e.g.

```
char c;
c=getch();
putch(c);
```

Output:

```
r    //output
char d;
d=getche();
putch(d);
```

Here getch() and getche() accepts entered key and putch() displays the pressed value.

gets() and puts():

gets() offer alternative function of scanf() function for reading strings. Similarly, puts() offer alternative function of printf() function for displaying the string. In general term, they are written as:

gets(string variable);

and puts(string variable);

Here, each function accepts a single argument. The argument must be a data item that represents a string. In case of get(), the string will be entered from keyboard and will terminate with a new line character (the string will end when the user presses the enter key).

Example:

```
#include<stdio.h>
void main()
{
char line[80];
gets(line);
puts(line);
```

This program transfer the line of text into the computer using gets() and display the line using puts().

**********END UNIT #5**********

Unit Six

Control Statements

The statements which alter the flow of execution of the program are called controlstatements. In the absence of control statements, the instruction or statements are executed inthe same order in which they appear in the program. Sometimes, we may want to executesome statements several times. Sometime we want to use a condition for executing only apart of program. So, control statements enable us to specify the order in which variousinstruction in the program are to be executed.

There are **two types of control statements**:

1. Loops : for, while, do-while

2. Decisions: if, if...else, nested if....else, switch

6.1 Loops

Loops are used when we want to execute a part of program or block of statementseveraltimes. So, a loop may be defined as a block of statements which are repeatedlyexecuted for a certain number of times or until a particular condition is satisfied. There arethree types of loopstatements in C:

1. For

2. While

3. Do...while

Each loop consists of two segments, one is k/a the control statement and the other is thebody of the loop. The control statement in loop decides whether the body is to be executed ornot. Depending on the position of control statement in the loop, loops may be classified eitherentry_controlled loop or exit_controlled loop. While and For are entry_controlled loopswhere as do...while is exit_controlled loop.

For Loop:

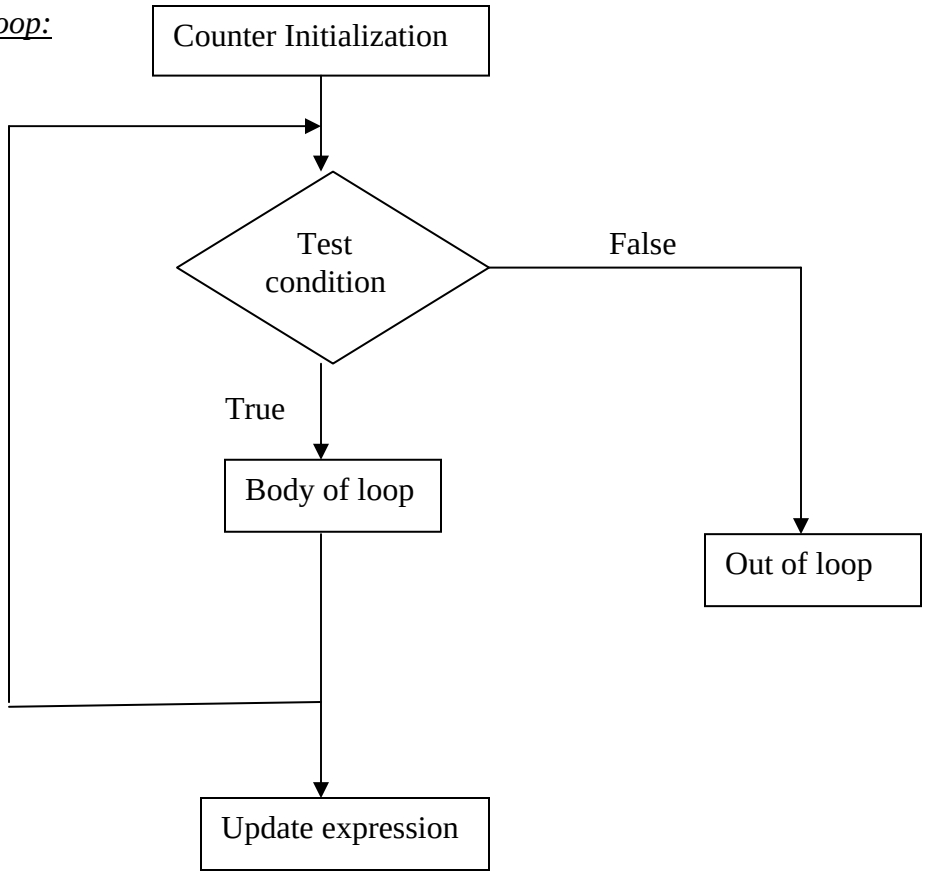
For loops is useful to execute a statement for a number of times. When the number ofrepetitions is known in advance, the use of this loop will be more efficient. Thus, this loop isalso known as determinate or definite loop.

The general syntax

```
for (counter initialization ; test condition ; increment or decrement)
{
| * body of loop * |
```

}

Flowchart of For Loop:



(Fig : Flowchart of For Loop)

For example:

```
|* Calculate the factorial of a number *|
#include<stdio.h>
main()
{
int num, i
long fact = 1
printf (“ln Enter a number whose factorial is to be calculated : “) ;
scanf (“%d”, &num) ;
for (i=1 ; i<=num ; i++)
fact * = i ; |* fact = fact*i *|
printf (“\n The factorial is : %d”, fact ) ;
}
```

Output:

While Loop:

The while statement can be written as

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

```
while(condition)
{
statement ; | * bed of the loop *|
statement ;
- - - - - - -
}
```

First the condition is evaluated; if it is true then the statements in the body of loop areexecuted. After the execution, again the condition is checked and if it is found to be true thenagain the statements in the body of loop are executed. This means that these statements areexecuted continuously till the condition is true and when it becomes false, the loop terminatesand the control comes out of the loop. Each execution of the loop body is called iteration.

```
|* Program to print the sum of digits of any num *|
#include<stdio.h>
main()
{
int n, sum = 0, rem ;
printf (“Enter the number :”) ;
scanf (%d”, &n) ;
while (n>0)
{
rem = n%10 ; |* taking last digit of number *|
sum+ = rem ; |* sum = sum + rem * |
n/ = 10 ; |* skipping last digit *|
```

do...while loop:

The do...while statement is also used for looping. The body of this loop may contain asingle statement or a block of statements. The general syntax is :

```
do
statement ;
while(condition) ;
```

```
do
{
Statement1 ;
Statement2 ;
- - - - - - - -
Statementn ;}
while(condition) ;
```

Here firstly the segments inside the loop body are executed and then thecondition is evaluated. If the condition is true, then again the loop body is executed and thisprocess continues until the condition becomes false. Unlike while loop, here a semicolon isplaced after the condition. In a 'while' loop, first the condition is evaluated and then thestatements are executed whereas in do while loop, first the statements are executed and thenthe condition is evaluated. So, if initially the condition is false the while loop will not executeat all, whereas the do while loop will always execute at least once.

```
|* program to print the number from 1 to 10 using do while *|
#include <stdio.h>
main()
{
int i = 1 ;
do
```

Differences between while loop and do while loop:

while

1. while loop is entry controlled loop i.e. testcondition is evaluated first and body of loopis executed only if this test is true.
2. The body of the loop may not be executedat all if the condition is not satisfied at thevery first attempt.
3. syntax :

while (condition)

{body of the loop}

4. Draw flowchart

do...while

1. do...while loop is exit controlled loop i.e.the body of the loop is executed first withoutchecking condition and at the end of body ofloop, the condition is evaluated.

2. The body of the loop is always executed atleast once.

3. syntax :

do

{

body of the loop

} while (condition)

4. Draw flowchart

Nesting of loops:

When a loop is written inside the body of another loop, then it is k/a nesting of loops.Any type of loop can be nested inside any other type of loop. For example, a for loop may benested inside another for loop or inside a while or do...while loop. Similarly, while and dowhile loops can be nested.

```
| * program to understand nesting in for loop * |
#include<stdio.h>
main()
{
int i,j ;
for (i = 1 ; i<=3 ; itt) |* Outer loop *|
{
printf (“i=%d|n”, i)
for (j=1 ; j<=4 ; j++) |* inner loop *|
printf (“j=%d\t”,j) ;
printf (“\n”) ;
}
}
```

Output:

```
i = 1
j = 1 j = 2 j = 3 j = 4
i = 2
j = 1 j = 2 j = 3 j = 4
i = 3
j = 1 j = 2 j = 3 j = 4
```

6.2 Decisions:

Since decision making statements control the flow of execution, they also fall under thecategory of control statements. Following are decision making statements:

1. if statements
2. if....else statements
3. else if statement
4. Nested if...else statement
5. switch statement

if statement:

The*if statement* is a powerful decision making statement and is used to control the flowof execution of statements. This is a bi-directional condition control statements. Thisstatement is used to test a condition and take one of two possible actions, If the condition istrue then a single statement or a block of statements is executed (one part of the program),other wise another single statement or a block of statements is executed (other part of theprogram). In C, any non-zero value is regarded as true while zero is regarded as false.

syntax :

if (condition) if (condition)

statement1 ; { statement1 ;

- - - - -

statement n ;

}

Here if the condition is true (non-zero) then statement1 is executed, and if it isfalse(zero), then the next statement which is immediately after the if control statement isexecuted.

For eg:

```
|* ### Program to check whether the number is –ve or +ve ### *|
#include<stdio.h>
main()
{
int num ;
printf (“Enter a number to be tested:” ) ;
scanf (“%d”, &num) ;
```

```
if (num<0)
printf("The number is negative") ;
printf ("value of num is : %d|n", num) ;
getch() ;
}
```

Output : 1st run

Enter a number to be tested : -6 ↵

The number is negative

Value of num is : -6

2nd run

Enter a number to be tested : 6 ↵

Value of num is : 6

if...else statement:

The if..else statement is an extension of the simple if statement. It is used when thereare two possible actions – one when a condition is true and the other when it is false.

Thesyntax is :

if (condition) if (condition)
statement1 ; {
else statement ;
statement2; } - - - -
else
{
statement ;
- - - -
}

Here if the condition is true then statement1 is executed and if it is false then statement2is executed. After this the control transfers to the next statement which is immediately afterthe if...else control statement.

|* Program to check whether the number is even or odd *|

```
#include<stdio.h>
main()
{
int num, remainder ;
printf (“Enter a number:”) ;
scanf (“%d”, &num) ;
remainder = num%2 ; |* modular division *|
if (remainder = = 0) |* test for even *|
printf (“Number is even|n”) ;
else
printf (“ Number is odd|n”) ;
}
```

Output:

Enter a number : 15 ↵

Number is odd.

Nested if ...else statement:

We can have another if... else statement in the if block or the else block. This is callednested if..else statement.

For example

```
if(condition1)
{
if(condition2)
statementA1;
else
statement A2;
        //Here, we have if...else inside both if block and else block
}
else
```

```
{
if(condition3)
statementB1;
else
statementB2;
}
```

|* Program to find whether a year is leap or not *|

```
#include<stdio.h>
main()
{
int year,
printf (“Enter year: ”) ;
scanf (“%d”, &year) ;
if(year%100 = = 0)
{
if(year%400 = = 0)
printf (“Leap year \n”) ;
else
printf (“Not leap year\n”) ;
}
```

This can also be written in place of nested if else as

```
if ((year%4 = = 0 && year %100!=0) | | year%400 = = 0)
printf (“%d is a leap year\n”, year) ;
else
printf (“%d is not a leap year\n”, year) ;
```

|* Program to find largest number from three given number *|

```
#include<stdio.h>
main()
{
int a, b, c, large ;
```

```
printf (“Enter three numbers : ”) ;
scanf (“%d%d%d”, &a, &b, &c) ;
if (a>b)
{
if (a>c)
large = a ;
else
large – c ;
}
else
{
if (b>c)
large = b ;
else
large = c ;
}
printf (“Largest number is %d\n”, large) ;
} /* End of main() */
```

Output:

Enter the numbers: 3 4 5 ↵

Largest num is 5

else if statement:

This is a type of nesting in which there is an if...else statement in every else part exceptthe last else part. This type of nesting is frequently used in programs and is also k/a else ifladder.

```
if(condition1) if(condition1)
statementA ; statementA ;
else elseif(condition2)
if(condition2) statementB ;
statementB ; elseif(condition3)
```

```
else statementC ;
if (condition3) else
statementC ; statement D ;
else
statement D ;
```

|* Program to find out the grade of a student when the marks of 4 subjects are given. The method of assuming grade is as
per>=80 grade = A
per<80 and per>=60 grade = B
per<60 and per>=50 grade = C
per<50 and per>=40 grade = D
per<40 grade = F
Here Per is percentage*|

```
#include<stdio.h>
main()
{
float m1, m2, m3, m4, total, per ;
char grade ;
printf (“Enter marks of 4 subjects : ”) ;
scanf (“%f%f%f%f”,&m1, &m2, &m3, &m4) ;
total = m1+m2+m3+m4 ;
per = total /4 ;
if(per>=80)
grade = ‘A’ ;
elseif(per>=60)
grade = ‘B’ ;
elseif(per>=50)
grade = ‘C’ ;
elseif(per>=40)
```

```
grade = ‘D’ ;
else
grade = ‘F’ ;
printf(“Percentage is %f\n Grade is %c\n”, per, grade) ;
}
```

Equivalent code in simple if statement:

```
if(per>=80)
grade = ‘A’ ;
if(per<80 && per>=60)
grade = ‘B’ ;
if(per<60 && per>=50)
grade = ‘D’ ;
if(per<40)
grade = ‘F’ ;
```

In else_if ladder whenever a condition is found true other conditions are not checked,while in if statement all the conditions will always be checked wasting a lot of time andmoreover the conditions here are more lengthy.

6.3 Statements: switch, break, continue, goto

Switch Statement:

This is a multi-directional conditional control statement. Sometimes, there is a need inprogram to make choice among number of alternatives. For making this choice, we use theswitch statement.

The general syntax is

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

```
- - - -
- - - -
case constantN:
statements ;
break,
default:
statements ;
}
```

Here, switch, case and default are keywords. The ‘expression’ following the switchkeyword can be any C expression that yields an integer value or a character value. It can bevalue of any integer or character variable, or a function call returning on integer, or arithmetic, logical, relational, bitwise expression yielding integer value.

The constants following the case keywords should be of integer or character type. Theseconstants must be different from each other.The statements under case can be any valid C statements like if...else, while, for or evenanother switch statement. Writing a switch statement inside another is called nesting ofswitches.

Firstly, the switch expression is evaluated then value of this expression is compared oneby one with every case constant. If the value of expression matches with any case constant,then all statements under that particular case are executed. If none of the case constantmatches with the value of the expression then the block of statements under default isexecuted.

```
|* Program to understand the switch control statment *|
#include<stdio.h>
main()
{ int choice ;
printf (“Enter your choice :” ) ;
scanf (“%d”, & choice)
switch(choice)
{
```

```
case 1 :
printf (“first|n”) ;
case 2 :
printf (“second|n”) ;
case 3 :
printf (“third|n”) ;
default :
printf (“wrong choice\n”) ;
}
```

Output:

```
Enter your choice : 2
Second
Third
Wrong Choice
```

Here value of choice matches with second case so all the statements after case 2 areexecuted sequentially. The statements of case 3 and default are also executed in addition tothe statements of case2. This is called falling through cases.Suppose we don’t want the control to fall through the statements of all the cases underthe matching case, then we can use break statement.

Break statement:

Break statement is used inside lops and switch statements. Sometimes it becomesnecessary to come out of the loop even before the loop condition becomes false. In such asituation, break statement is used to terminate the loop. This statement causes an immediateexit from that loop in which this statement appears. It can be written as (i.e. general syntax) :

break,

If a break statement is encountered inside a switch, then all the statements followingbreak are not executed and the control jumps out of the switch.

/* Program to understand the switch with break statement */

```
#include<stdio.h>
main()
{
int choice ;
printf (“Enter your choice : ”) ;
scanf (“%d”, & choice) ;
switch (choice)
{
case1:
print (“First\n”) ;
break ; /* break statement */
case2:
printf(“Second\n”) ;
break ;
case3:
printf (Third\n”) ;
break ;
default :
printf (“Wrong choice\n”) ;
}
} /* End of main() */
```

Output:

Enter your choice : 2 ↵

/* Program to perform arithmetic calculation on integers */

```
#include<stdio.h>
main()
{ char op ;
int a, b ;
printf (“Enter a number, operators and another num :”) ;
scanf (“%d%c%d, &a, &op, &b) ;
```

```
switch (op)
{
case ‘+’ :
printf (“Result = %d\n”, a+b) ;
break ;
case ‘-‘ :
printf (“Result = %d\n”, a-b) ;
case ‘*’ :
printf (“Result = %d\n”, a*b) ;
case ‘/’
printf (“Result = %d\n”, a/b) ;
case ‘%’
printf (“Result = %d\n”, a%b) ;
default:
printf (“Enter your valid operation”) ;
} /* end of switch */
} /* end of main() */
```

/* Program to find whether the alphabet is a vowel or consonant */

```
#include<stdio.h>
main()
{ char ch ;
printf (“Enter an alphabet :” ) ;
scanf (“%c”, &ch) ;
switch (ch)
{
case ‘a’ : case ‘A’ :
case ‘e’ : case ‘E’ :
```

```
case ‘i’ : case ‘I’ :
case ‘o’ : case ‘O’ :
case ‘u’ : case ‘U’ :
printf (“Alphabet is a vowel\n”) ;
break ;
default :
printf (“Alphabet is a constant\n”) ;
} /* end of switch */
} * end of main() */
```

Continue statement:

The continue statement is used to bypass the remainder of the current pass through aloop. The loop does not terminate when a continue statement is encountered. Instead theremaining loop statements are skipped and the computation proceeds directly to the next passthrough the loop.

The general syntax :

continue ;

/* Program to demonstrate continue statement */

```
#include<stdio.h>
main()
{
int i, num ;
printf (“\n Enter a number :”) ;
scanf (%d”, &num) ;
printf (“\n The even numbers from 2 to %d are : \n”, num) ;
for (i=1, ; i<=num ; i++)
{ if(i%2!=0)
```

```
continue ;
printf (“\t%d”, i) ;
{ /* end of for loop */
} / * end of main() */
```

Output:

```
Enter a number : 20
The even numbers from 2 to 20 are
        1    4 6 8 10 12 14 16 18 20
```

goto statement:

The goto statement is used to alter the normal sequence of program execution byunconditionally transferring control to some other part of the program. The goto statementtransfers the control to the labeled statement somewhere in the current function.

The generalsyntax of goto statement:

goto label ;

- - - - -

- - - - -

label :

statement ;

- - - - -

- - - - -

Here, label is any valid C identifier and it is followed by a colon. Whenever, the statementgoto label, is encountered, the control is transferred to the statement that is immediately afterthe label.

Generally, the use of goto statement is avoided as it makes program illegible and unreliable.This statement is used in unique situations like

• Branching around statements or group of statements under certain conditions

• Jumping to the end of a loop under certain conditions, thus bypassing the remainder ofloop during current pass.

• Jumping completely out of the loop under certain conditions, terminating the executionof a loop.

/* Program to print whether the number is even or odd */

```
#include<stdio.h>
main()
{int n ;
printf( “Enter the number :”) ;
scanf (“%d”, &n) ;
if (n%2 = = 0)
goto even ;
else
goto odd;
even :
printf (“Number is even”) ;
goto end ;
odd :
printf (“Number is odd”) ;
end :
printf (“\n) ;
}
```

Output:

Enter the number : 6 ↵

Number is even.

6.4 Exit( ) function:

We have already known that we can jump out of a loop using either the break statementor goto statement. In a similar way, we can jump out of a program by using the libraryfunction exit( ). In case, due to some reason, we wish to break out of a program and return tothe operating system.

The general syntax is

- - - - -

- - - - -

if (condition) exit (0) ;

- - - - -

- - - - -

The exit( ) function takes an integer value as its argument. Normally zero is used toindicate normal termination and non zero value to indicate termination due to some error orabnormal condition. The use of exit( ) function requires the inclusion of the header file<stdio.h>.

/* Program to demonstrate exit( ) */

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int choice ;
while(1)
{
printf (“ 1. Create database\n”) ;
printf (“ 2. Insert new record\n”) ;
printf (“ 3. Modify a record\n”) ;
printf (“ 4. Delete a record\n”) ;
printf (“ 5. Display all records\n”) ;
printf (“ 6. Exit\n”) ;
printf (“Enter your choice :” ) ;
scanf (“%d”, &choice) ;
switch (choice)
{
case1:
printf (“:datase created - - -\n\n”) ;
break ;
```

```
case2:
printf (“Record inserted - - -\n\n”) ;
break ;
case3:
printf (“Record modified - - -\n\n”) ;
break ;
case4:
printf (“Record deleted - - -\n\n”) ;
break ;
case5:
printf (“Record displayed - - -\n\n”) ;
break ;
case6:
exit(1)
default:
printf (“Wrong choice\n”) ;
} /* end of switch */
} /* end of while */
} /* end of main() */
```

**********END UNIT #6**********

Unit Seven

Arrays and Strings

Array is a data structure that store a number of data items as a single entity (object). The individual data items are called elements and all of them have same data types. Array is used when multiple data items that have common characteristics are required (in such situation use of more variables is not efficient). In array system, an array represents multiple data items but they share same name. The individual data items can be characters, integers,

floating point numbers etc. However, they must all be of the same type and the same storage class.

Declaration of array

Syntax:

storage_class data_type array_name[expression];

Where, **storage_class** refers to the storage class of the array. It may be auto, static, extern. But it is optional.

data_type is the data type of array. It may be int, float, char, ….etc. If int is used, this means that this array stores data items of integer types.

array_name is name of the array. It is user defined name for array.

[] represents the array and **expression** inside this may be integer constant like 10, 100 or Symbolic constant like SIZE (i.e. *#define SIZE 80, this can be used as int a[size];*). This expression represents the number of elements in array.

Example:

int a [5];

It tells to the compiler that 'a' is an integer type of array and can store 5 integers. The compiler reserves 2 bytes of memory for each integer array element.

It's individual elements are recognized by *a[0], a[1],a[2],a[3]* and *a[4]*. The integer value inside [] is called index of array. Index always starts from 0 and ends with one less than size of array.

Similarly, *long num[5]; char ch[10];* are another examples of array declarations.

Initialization of array:

Till the array elements are not given any specific values, they are supposed to contain garbage value. The values can be assigned to elements at the time of array declaration, which is called array initialization. The syntax is

Storage_class data_type array_name[expression]={vaue1, value2, …value n};

Where, value1 is value of first element, value2 is that of second and so on.

The array initialization is done as given below:

```
int a[5]= {1,2,3,5,8};
 int b[]= {2,5,7};
 int c[10]={45,89,54,8,9};
```

In first example, a is integer type array which has 5 elements. Their values are assigned as 1, 2, 3, 5, 8 (i.e. a[0]=1, a[1]=2,a[2]=3,a[3]=5 and a[4])=8. The array elements are stored sequentially in separate locations.

In second example, size of array is not given, it automatically its size as 3 as 3 values are assigned at the time of initialization. Thus, writing int b[]= {2,5,7}; and int b[3]= {2,5,7}; is same.

In third example, the size of array c has been set 10 but only 5 elements are assigned at the time of initialization. In this situation, all individual elements that are not assigned explicit initial values will automatically be set to zero. Thus, the value of c[5]=0, c[6]=0 …… c[9]=0.

Array input and output:

Loops are used for array input/outputs. To access the individual elements of array, we use subscripts as a[subscript], where subscript could be a constant integer like 0 , 1, 5 or a integer variable like i, j or a integer expression like (i+j+2). One important thing to note is that the value of subscript should be integer.

e.g.

```
#include<stdio.h>
#include<conio.h>
void main()
 {
int a[10], i;
clrscr();
printf("Enter 10 numbers:\n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);  /* array input */

printf("\n You have entered these 10 numbers:\n");
for(i=0;i<10;i++)
printf("\t%d",a[i]);  /* array output*/
```

This example shows how to input array elements by users and output them. This program asks 10 integers from user and then displays these numbers.

Characteristics of Array

- The declaration int a [5] is nothing but creation of 5 variables of integer types in the memory. Instead of declaring five variables for five values, the programmer can define them in the array.
- All the elements of an array share the same name, and they are distinguished from one another with the help of an element number.
- The element number in the array plays an important role for calling each element.
- Any particular element of an array can be modified separately without disturbing other elements.
- Any element of an array a[ ] can be assigned/equated to another ordinary variable or array variable of its type.

```
e.g. int b, a[10];
   b=a[5];
a[2]=a[3];
```

One dimensional array

One dimensional array can be thought as a list of values. In one dimensional array, there is only one subscript like num[10], where 10 is called subscript or index.

e.g. int a[5];

1th element a[0]	2nd element a[1]	3rd element a[2]	4th element a[3]	5th element a[4]	200
0	2002	2004	2006	2008	

The elements of an integer array a[5] are stored in continuous memory locations. It is assumed that the starting memory location is 2000.As each integer element requires 2 bytes, subsequent element appears after gap of 2 locations.

Multi-dimensional arrays

Multi-dimensional arrays are those which have more than one dimensions. Multi-dimensional arrays are defined in much the same manner as one dimensional array, except that a separate pair of square brackets is required for each subscript. Thus, two dimensional arrays will require two pairs of square brackets; three dimensional arrays will require three pairs of square brackets and so on. The two dimensional array is also called matrix. Multi-dimensional array can be defined as following

storage_class data_type array_name[expression1][expression2]...[expression n];

Here, expression1, expression2…..expression n are positive valued integer expressions that indicate the number of array elements associated with each subscript. An m*n two dimensional array can be thought as tables of values having m rows and n columns.

An example

int X[3][3] can be shown as follows

	Col 1	Col 2	Col 3
Row 1	X[0][0]	X[0][1]	X[0][2]
Row 2	X[1][0]	X[1][1]	X[1][2]
Row 3	X[2][0]	X[2][1]	X[2][2]

Initialization of multi-dimensional array:

Like one dimensional array, multi-dimensional arrays can be initialized at the time of array definition or declaration. A few example of 2-D array initialization are:

```
    int marks[2][3]={   {2, 4, 6},
                        {8, 10, 12}
                    };
int marks1[][3]={   {2, 4, 6},
                        {8, 10, 12}
                    };
    int marks2[2][3]={ 2,4,6,8, 10, 12};
    int marks3[][3]={ 2,4,6,8, 10, 12};
int marks4[][3]={ 2,4,6,8, 10};

What is the value of marks4[1][2]?
```

All above examples are valid. While initializing an array, it is necessary to mention the second dimension (column) whereas the first dimension (row) is optional.

Thus, following are invalid

```
int marks3[3][]={ 2,4,6,8, 10, 12};
int marks4[][]={ 2,4,6,8, 10};    /* error!!!*/
```

Example: Write a program to read two 3*3 matrixes and display their sum.

```
#include<stdio.h>
#include<conio.h>
void main()
 {
 int a[3][3],b[3][3],sum[3][3],i,j;
clrscr();
printf("Enter First matrix row by row\n");
for(i=0;i<3;i++)
for(j=0;j<3;j++)
scanf("%d",&a[i][j]);

printf("Enter second matrix as first\n");
for(i=0;i<3;i++)
for(j=0;j<3;j++)
    scanf("%d",&b[i][j]);
```

Assignment:

***Write a program to read two matrixes suitable for multiplication and display their product matrix*.**

Processing an array:

Single operations which involve entire arrays are not permitted in C. The assignment operations, comparison operations etc can not be carried in entire array; instead it is carried out on an element by element basis. These operations are usually accomplished using loop which is clear above matrix and input/output examples..

Passing arrays to function

An entire array can be passed to a function as an argument. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declaration. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

Syntax for function call passing array as argument,

Function_name(array_name);

Syntax for function prototype which accepts array

Return_type function_name (data_type array_name[]);

Example:

Write a program to read 10 numbers and reorders them in ascending order using function.

```
void ascending(int num[])
  {
int i,j,temp;
for(i=0;i<9;i++)
    for(j=i+1;j<10;j++)
     {
    if(num[i]>num[j])
     {
    temp=num[i];
    num[i]=num[j];
    num[j]=temp;
      }
     }
    }
```

When an array is passed to a function, the values of the array elements are not passed to the function. Rather, the array name is interpreted as the address of the first array element. This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first array element. Thus, array is passed using call by reference. Thus, changing the values in function is also recognized in main function.

An example:

```
void change(int a[])
  {
     a[0]=10;  a[1]=20;     a[2]=30;     a[3]=40;     a[4]=50;
   }
void main()
 {
int nums[5]={1,2,3,4,5},i;
clrscr();
printf("Before function call\n");
for(i=0;i<5;i++)
    { printf("\t%d",nums[i]); }
```

Output:-

```
Before function call
    1    2    3    4    5
After function call
    10   20   30   40   50
```

Arrays and strings

It is the fact that a string can be represented as a one-dimensional character -type array. Thus, character array is also known as string. Each character within the string will be stored within one element of the array. Strings are used by C to manipulate text such as words and sentences. A string is always terminated by a null character (i.e \0). The terminating null character \0 is important because it is the only way the string handling functions can know where the string ends.

An ex ample:

```
void main()
 {
char line[40];
int i=0;
clrscr();
printf("Enter text\n");
gets(line);
printf("\nYour text in discrete form\n");

            while(line[i]!='\0')
             {
            printf("\t%c",line[i]);
            i++;
             }

getch();
 }
```

This example checks the end of string using terminating character '\0'. C inserts the null character automatically at the end of string. Thus, it is not necessary to enter this character by the user.

String Handling Functions

The library or built-in functions strlen(), strcpy(), strcat(), strcmp(), strrev() etc are used for string manipulation in C. These functions are defined in header file string.h. They are described in detail as follow.

i. strlen()

This function returns the length of string. Its prototypes is as

int strlen(string s1);

example:

```
#include<string.h>
void main()
 {
char line[]="baba";
int len;
clrscr();
len=strlen(line);
printf("The length is %d",len);
getch();
  }

output:
 The length is 4
```

ii. strcpy()

This function is used to copy one string to another. It is called as

strcpy(destination_string, source_string);

Thus, strcpy(s1,s2) means the content of s2 is copied to s1.

An example:

```
#include<string.h>
void main()
 {
char s1[]="adhikarib", s2[]="bindu";
clrscr();
strcpy(s1,s2);
printf("The string s1 is\t%s",s1);
getch();
 }

output:
```

iii. strcat()

This function concatenates the source string at the end of the destination or target string.

An example

```
#include<string.h>
void main()
 {
char s1[]="Bindu", s2[]="Adhikari";
clrscr();
strcat(s1,s2);
printf("The concated string is:\t%s",s1);
getch();
 }
```

Output:

```
The concated string is:BinduAdhikari
```

iv. strcmp();

This function compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of stings is reached. It two strings are equal, it returns zero, otherwise, it returns the difference between the ASCII values of the first non-matching pair of characters.

An example

```
#include<string.h>
void main()
 {
char s1[]="bAa", s2[]="eaA";
int diff;
clrscr();
diff= strcmp(s1,s2);
```

v. strrev();

This function is used to reverse all characters in a string except null character at the end of string. The reverse of string "abc" is "cba".

```
#include<string.h>
void main()
 {
char s[]="cps";
clrscr();
strrev(s);
printf("%s",s);
getch();
 }
```

Output: spc

Examples of sing

/* Program that reads 10 integers from keyboard and displays entered numbers in the screen*/

```
#include<stdio.h>
main( )
{
```

```
int a[10], i ;
clrscr( ) ;
printf (“Enter 10 numbers : \t”) ;
for (i=0 ; i<10 ; i++)
scanf(“%d”, &a[i]) ; /*array input */
printf (“\n we have entered these 10 numbers : \n”) ;
for (i=0 ; i<10 ; i++)
printf (“\ta[%d]=%d”, i, a[i] ) ; /* array o/p */
getch( ) ;
}
```

Output:

Enter 10 numbers : 10 30 45 23 45 68 90 78 34 32

We have entered these 10 numbers:

a[0] = 10 a[1] = 30 - - - - - - - - - - - - - - - - -a[9] = 32

/* Program to illustrate the memory locations allocated by each array elements */

```
#include <stdio.h>
main( )
{
float a[ ] = {10.4, 45.9, 5.5, 0, 10.6}
int i ;
clrscr( ) ;
printf (“The continuous memory locations are : \n” ) ;
for (i=0; i<5; i++)
printf( “\t%u”, &a[i]) ; /*address of array element */
getch( ) ;
}
```

Output:

The continuous memory locations are

65506 65510 65514 65514 65522

```
/* Program to sort n numbers in ascending order */
main( )
{int num[50], i, j, n, temp,
clrscr( ) ;
printf("How many numbers are there? \t") ;
scanf("%d", &n) ;
printf("\n Enter %d numbers: \n", n) ;
for (i=0; i<n; i++)
{
for (j=i+1 ; j<n ; j++)
{
if(num[i]>num[j]) ;
{
temp = num[i] ;
num[i] = num[j] ;
num[j] = temp ;
} /* end of if */
} /*end of inner loop j*/
} /* end of outer loop i */
printf("\n The numbers in ascending order : \n") ;
for (i=0 ; i<n ; i++)
printf("\t%d, num[i] ) ;
getch( );
} /* end of main */
```

Output:

How many numbers are there? 5

Enter 5 numbers: 12 56 3 9 17

The numbers in ascending order: 3 9 12 17 56

Examples of Multi-dimensional arrays:

/* Program to read & display 2×3 matrix */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int matrix [2] [3], i, j ;
clrscr( ) ;
for (i=0 ; i<2 ; i++)
{
        for (j = 0 ; j<3 ; j++)
        {
        printf("Enter matrix [%d] [%d] : \t", i, j) ;
        scanf("%d", &matrix [i] [j] ) ;
        }
}
printf ("\n Entered matrix is : \n") ;
for (i=0 ; i<2 ; i++)
{
printf("%d\t", matrix [i] [j]) ;
printf("\n") ;
}
getch( )
}
```

Output:

```
Enter matrix [0] [0] : u1
Enter matrix [0] [1] : u2
Enter matrix [0] [2] : u3
Enter matrix [1] [0] : u4
Enter matrix [1] [1] : u5
Enter matrix [1] [2] : u6
```

Entered matrix is

```
1   2   3
4   5   6
```

/* Program to read two matrices and display their sum */

```
#include<stdio.h>
#include<conio.h>
main( )
{
int a[3] [3], b[3] [3], s[3] [3], i, j ;
clrscr( ) ;
printf("Enter first matrix : \n") ;
for (i=0 ; i<3 ; i++)
for (j=0 ; j<3 ; j++)
scanf("%d", &a[i] [j]) ;
printf("Enter second matrix : \n") ;
for (i=0 ; i<3 ; i++)
for (j=0 ; j<3 ; j++)
scanf("%d", &b[i] [j]) ;
for (i=0 ; i<3 ; i++)
for (j=0 ; j<3 ; j++)
s[i] [j] = a[i] [j] + b[i] [j] ;
printf("\n The sum matrix is : \n") ;
for (i=0 ; i<3, i++)
{
        for(j=0 ; j<3 ; j++)
        {
        printf("\t%d", s[i] [j]) ;
        }
}
```

```
getch( ) ;
}
```

Output:

```
Enter first matrix:
1       2       3
4       5       6
7       8       9
Enter second matrix:
9       8       7
6       5       4
3       2       1
The sum matrix is
10      10      10
10      10      10
10      10      10
```

/* Program to read two matrices and multiply them if possible */

```
#include<stdio.h>
#include<conio.h>
main( )
{
int a[10] [10], b[10] [10], s[10] [10] ;
int m, n, l, p, i, j, k ;
 printf("Enter row of first matrix(<=10) : \t") ;
scanf("%d", &m) ;
printf("Enter column of first matrix(<=10) : \t") ;
scanf("%d", &n) ;
printf("Enter row of second matrix(<=10) : \t") ;
scanf("%d", &l) ;
printf("Enter column of second matrix(<=10) : \t") ;
scanf("%d", &p) ;
```

```
if (n!=l)
printf(“Multiplication is not possible :”) ;
else
{
printf(“Enter the first matrix : \n”) ;
for (i=0 ; i<=m-1 ; i++)
{
for (j=0 ; j<=n-1 ; j++)
{
printf(“Enter a[%d] [%d] : \t”, i, j) ;
scanf (“%d”, &a [i] [j] ;
}
}
printf(“Enter the second matrix : \n”) ;
for (i=0 ; i<=l-1 ; i++)
{
for (j=0 ; j<=p-1 ; j++)
{
printf (“Enter b[%d] [%d] :\t”, i, j) ;
scanf(“%d”, &b[i] [j] );
}
}
for (i=0 ; i<=m-1 ; i++)
for (j=0 ; j<=p-1 ; j++)
s[i] [j] = 0 ;
for (i=0 ; i<=m-1 ; i++)
for (j=0 ; j<=p-1 ; j++)
for (k=0 ; k<=n-1 ; k++)
s[i] [j] = s[i] [j] + a[i] [k] * b[k] [j] ;
printf (“The matrix multiplication is : \n”) ;
for (i=0 ; i<=m-1 ; i++)
```

```
{
for (j=0 ; j<=p-1 ; j++)
{
printf("%d\t", s[i] [j],
}
printf("\n") ;
}
} /* end of else */
getch( ) ;
} /* end of main( ) */
```

Output:

```
Enter row of the first matrix (<=10) : 2
Enter column of the first matix (<=10) : 1
Enter row of the second matrix (<=10) : 1
Enter column of the second matrix (<=10) : 2

Enter the first matrix:
Enter a[0] [0] : 2
Enter a[1] [0] : 2

Enter the second matrix:
Enter b[0] [0] : 3
Enter b[0] [1] : 3
The matrix multiplication is :
6       6
6       6
```

Passing arrays to functions:

Like any other variables, we can also pass entire array to a function. An array name canbe named as an argument for the prototype declaration and in function header. When we callthe function no need to subscript or square brackets. When we pass array that pass as a callby reference because the array name is address for that array.

/* Program to illustrate passing array to function */

```
#include<stdio.h>
void display(int) ; /* function prototype */
main( )
{
int num[5] = {100, 20, 40, 15, 33, i ;
clrscr( ) ;
printf (“\n The content of array is \n”) ;
for (i=0; i<5; i++)
display (num[i]) ; /*Pass array element fo fun */
getch{ } ;
}
void display(int n)
{
printf (“\t%d”, n ) ;
}
```

Output:

```
The content of array is
100      20       40       15 3
```

/* Program to read 10 numbers from keyboard to store these num into array and thencalculate sum of these num using function */

```
#include<stdio.h>
#include<conio.h>
int sum (int a[]) ;
void output (int a[]) ;
```

```
main( )
{
int a[10], s, i ;
clrscr( ) ;
printf (“Enter 10 elements : \n”) ;
for (i=0 ; i<=9 ; i++) ;
scanf (“%d”, &a[i]) ;
output (a) ;
s = sum (a) ;
printf (“Sum of array element is :\n”, s) ;
getch( ) ;
}
int sum (int a[ ] )
for (i=0 ; i<=9 ; i++)
n = n+a[i] ;
return (n) ;
}
void output (int a[])
{
int i;
printf (“The elements of array are : \n”) ;
for (i=0 ; i<=9 ; i++) ;
printf (“%d\n”, a[i]) ;
}
```

Output:

```
Enter 10 elements:
5          12
15         8
10         20
2          30
```

8 40

The elements of array are:

5

15

10

2

8

12

8

20

30

40

Sum of array element is : 150

/* Passing two dimensional array as an argument to function */

/* Program to read two matrices A and B of order 2×2 and subtract B from A using function*/

```
#include<stdio.h>
#include<conio.h>
void input (int t[ ] [2]) ;
void output (int t1[ ] [2]) , int t2[ ] [2] ) ;
main( )
{
int a[2] [2], b[2] [2] ;
clrscr ( ) ;
printf (“Enter first array :\n” ) ;
input(a)
printf (“Enter second array :\n”) ;
output(a,b) ;
getch( ) ;
}
```

```
void input (int t[ ] [2])
{
int i,j ;
for (i=0 ; i<=1 ; i++)
for (j=0 ; j<=1 ; j++)
scanf (“%d”, &t [ i] [j]) ;
}
void output (int t1[ ] [2], int t2[ ] [2])
{
int i,j ;
int s[2] [ 2] ;
for (i=0, i<=1 ; i++)
for (j=0 ; j<=1 ; j++)
s[i] [j] = t1[i] [j] – t2[i] [j] ;
for (i=0 ; i<=1 ; i++)
{
for ( ) = 0 ; j<=1 ; j++)
printf(“%d\t”, s[i] [j]) ;
printf(“\n”) ;
}
}
```

Output:

Enter first array:

4

3

2

4

Enter second array:

1

2

3

4

Subtraction is:

3 1

-1 2

/* Program to illustrate string initialization */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
char name[ ] = “Ram” ;
clrscr( ) ;
printf (“Your name is :%s\n”, name)
getch( ) ;
}
```

Output:

Your name is:

Ram

/* A program to read a string and check for palindrome */

```
void main( )
{
char st[40] ;
int len i, pal = 1 ;
clrscr( ) ;
printf (“Enter string of our choice:”) ;
gets(st) ;
len = strlen(st) ;
for (i=0; i<(len 12); i++)
{
if (st[i] ! = st[len-i-1])
```

```
pal = 0 ;
}
if (pal = = 0)
printf (“\n The input string is not palindrome”) ;
else
printf (“\n the input string is palindrome”) ;
getch( ) ;
}
```

**********END UNIT #7**********

Unit Eight

Functions

A function is defined as a self contained block of statements that performs a particular task. This is a logical unit composed of a number of statements grouped into a single unit. It can also be defined as a section of a program performing a specific task. [Every C program can be

though of as a collection of these functions.] Each program has one or more functions. The function main(0 is always present in each program which is executed first and other functions are optional.

Advantages of using Functions:

The advantages of using functions are as follows:

1. Generally a difficult problem is divided into sub problems and then solved. This divideand conquer technique is implemented in C through functions. A program can be dividedinto functions, each of which performs some specific task. So, the use of C functionsmodularizes and divides the work of a program.
2. When some specific code is to be used more than once and at different places in theprogram, the use of function avoids repetition of that code.
3. The program becomes easily understandable. It becomes simple to write the program andunderstand what work is done by each part of the program.
4. Functions can be stored in a library and reusability can be achieved.

Types of Functions

C program has two types of functions:

1. Library Functions
2. User defined functions

Library Functions:

These are the functions which are already written, compiled and placed in C Libraryand they are not required to be written by a programmer. The function's name, its return type,their argumentnumber and types have been already defined. We can use these functions asrequired. For example: printf(), scanf(), sqrt(), getch(), etc.

User defined Functions:

These are the functions which are defined by user at the time of writing a program. Theuser has choice to choose its name, return type, arguments and their types. The job of eachuser defined function is as defined by the user. A complex C program can be divided into anumber of user defined functions.

For example:

```
#inlcude<stdio.h>
double convert (int) ; /* function proto type */
main()
{ int c ;
/* double d ; */
printf (“Enter temperature in Celsius: ”) ;
scanf (“%d”, &c) ;
/* d = convert(c) ; */ /*Function call*/
printf (“The Fahrenheit temperature of %d C = $lf F”;c, convert (c) ;
/* d inplace of convert */
}
double convert (int C) /* function definition */
{
double f ;
f = 9.0*c/5.0+32.0 ;
return f ;
}
```

What is about main( ) function?

The function main() is an user defined function except that the name of function isdefined or fixed by the language. The return type, argument and body of the function aredefined by the programmer as required. This function is executed first, when the programstarts execution.

Function Declaration or Prototype:

The function declaration or prototype is model or blueprint of the function. If functionsare used before they are defined, then function declaration or prototype is necessary. Manyprogrammers prefer a “top-down” approach in which main appears ahead of the programmerdefined function definition. Function prototypes are usually written at the beginning of aprogram, ahead of any user-defined function including main(). Function prototypes providethe following information to the compiler.

- The name of the function
- The type of the value returned by the function
- The number and the type of arguments that must be supplied while calling thefunction.

In “bottom-up” approach where user-defined functions are defined ahead of main()function, there is no need of function prototypes. The general syntax of function prototype is

return_type function_name (type1, type2, ..., typen) ;

where, return_type specifies the data type of the value returned by the function. Afunction can return value of any data type. If there is no return value, the keyword void isused. type1, type2, ..., typen are type of arguments. Arguments are optional.

For example:

```
int add (int, int) ; /* int add (int a, int b) ;
void display (int a) ; /* void display (int); */
```

Function definition:

A function definition is a group of statements that is executed when it is called fromsome point of the program. The general syntax is

```
return_type function_name (parameter1, parameter2, ....., parametern)
{ - - - - -
statements ;
- - - - -
}
```

Where,

- return_type is the data type specifier of data returned by the function.
- function_name is the identifier by which it will be possible to call the function.

Parameters (as many as needed) : Each parameter consists of a data type specifierfollowed by an identifier like any regular variable declaration. (for eg: int x) andwhich acts within the function as a regular local variable. They allow to passarguments to the function when it is called. The different parameters are separatedby commas.

Statements is the function's body. It is a block of statements surrounded bybraces{}.

The first line of the function definition is k/a function header.

```
#include<stdio.h>
int addition (int a, int b) /* function prototype */
main()
{ int 2 ;
2 = addition (5,3) ; /*function call */
printf ("The result is%d"z) ;
}
int addition(int a, int b) /* function header */
{ int r;
r = a+b
return r;
}
```

Output:

The result is 8

Return Statement:

It is the statement that is executed just before the function completes its job and control is transferred back to the calling function. The return statement serves mainly two purposes. They are:

1) It immediately transfers the control back to the calling program after executing the return statement.

2) It returns the value present in the parentheses to the calling function.

The syntax for return is:

return expression;

Here, the value of expression is returned to the calling portion of the program. The expression is optional. If the expression is omitted, the return statement simply causes the control to revert back to the calling portion of the program without any transfer of information. A function definition can include multiple return statements, each containing a different expression. But, a function can return only one value to the calling portion of the program via return.

/* Program to understand the use of return statement */

```
#include<stdio.h>
void funct( int, float) ;
main()
{ int age ;
float ht ;
printf (“Enter age and height :”) ;
scanf (“%d%f”, &age, &ht) ;
funct (age, ht) ;
}
void funct (int age, float ht)
{
if (age>25)
{ printf (“Age should be less than 25\n”) ;
return ;
}
if (ht<5)
{ printf (“Height should be more than 5\n”
return ; }
print (“selected \n”) ;
}
```

The second form of return statement is used to terminate a function and return a value to the calling function. The value returned by the return statement may be any constant, variable, expression or even any other function call which returns a value.

For example:

return 1 ;

return x++

return (x+y*z)

return (3*sum(a,b)) ;

Accessing a function:

A function can be called or accessed by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. For example, add(a,b) to call a function to add two numbers. If function call doesn't require any arguments, any empty pair of parentheses must follow the name of function. The arguments appearing in the function call are referred as actual arguments. In function call, there will be one argument for each formal argument. The value of each argument is transferred into the function and assigned to the corresponding formal argument.

An example

```
/* to find the greatest number among three numbers.*/
    #include <stdio.h>
    #include<conio.h>
     int greater(int x, int y)    /* function definition*/
        {
          if(x>y)
               return (x);
          else
             return (y);
         }

 void main()
         {
```

```
    int a, b,c,d,e;
    clrscr();
    printf("Enter three numbers");
    scanf("%d%d%d",&a,&b,&c);
    d=greater(a,b);        /* function call*/
    e=greater(d,c);        /* same function call again*/
    printf("The greatest number is%d",e);
    getch();
   }
```

Function call by value:

When actual value of variable is passed to the function as argument, it is known as function call by value. In this call, the value of the actual argument is copied into the function's formal argument. Therefore, the value of the corresponding formal argument can be altered within the function. But, this will not change the value of the actual argument within the calling function.

Example:

```
    #include<stdio.h>
    #include<conio.h>
    void swap(int , int);
      void main()
        {
          int a, b;
          a=99; b=89;
          printf("Before function calling, a and b are: %d\t%d",a,b);
          swap(a,b);
          printf("After function calling, a and b are: %d\t%d",a,b);
getch();
         }

void swap(int x, int y)
      {
```

```
        int temp;
        temp=x;
        x=y;
y=temp;
        printf("The values within functions are:%d\t%d",x,y);
       }
```

Output:

```
Before function calling, a and b are: 99        89
    The values within functions are: 89         99
    After function calling, a and b are: 99     89
```

The above output shows that the actual argument's value are not changed even the value of formal arguments are changed.

/* Program to illustrate the function with argument and return values */

```
int add(int, int) ;
main()
{
int a, b, sum ;
printf ("Enter two numbers : \t") ;
scanf ("%d%d", &a, &b) ;
sum = add(a, b) ;
printf ("ln the sum is \t%d|, sum) ;
}
int add(int a, int b)
{
int sum ;
sum = a+b ;
return sum ;
}
```

Output:

Enter two numbers : 4 3 ↵

The sum is 7

Function call by Reference:

In this type of function call, the address of variable is passed to the function as argument instead of actual value of variable. For clear understanding of this, concept of pointer is necessary which we will study in next chapter.

Example:

```
/*a program to swap the two variable's values*/
#include<stdio.h>
#include<conio.h>
void swap(int * , int *);
  void main()
    {
      int a, b;
      a=99; b=89;
      printf("Before swap, a and b are: %d\t%d",a,b);
      swap(&a,&b);          /*  function call by reference*/
      printf("After swap, a and b are: %d\t%d",a,b);
      getch();
     }
void swap(int *x, int *y)
    {
      int temp;
      temp=*x;  /*  here *x represent the value at address contained in variable x */
*x=*y;
      *y=temp;
      }
```

Output:

Before swap, a and b are: 99 89

After swap, a and b are: 89 99

Recursive Function

If a statement within the body of a function calls the same function, the function is called recursive function. Actually, recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. This process is used for repetitive computations in which each action is stated in term of previous result. Many iterative or repetitive problems can be written in this form.

To solve a problem using recursive method, two conditions must be satisfied. They are:

1) Problem could be written or defined in term of its previous result.
2) Problem statement must include a stopping condition.

```
/*  An example of recursive function to calculate factorial of a number.*/
  #include<stdio.h>
  #include<conio.h>
   long int factorial(int n)
        {
          if(n==1)
              return(1);
          else
               return (n*factorial(n-1));
          }

void main()
       {
         int num;
         printf("Enter a number:");
         scanf("%d",&num);
           printf("The factorial is %ld", factorial(num));
           getch();
        }
```

Difference between Recursion & Iteration:

Recursion

1. A function is called from the definition of the same function to do repeated task.
2. Recursive is a top-down approach to problem solving; it divides the problem into pieces.
3. In recursion, a function calls to itself until some condition will be satisfied.
4. Problem could be defined in terms of its previous result to solve a problem using recursion.
5. All problems cannot be solved using recursion.

Iteration

1. Loop is used to do repeated task.
2. Iteration is like bottom-up approach, it begins with what is known and from this it contacts the solution step by step.
3. In iteration, a function doesn't call to itself.
4. It is not necessary to define a problem in term of its previous result to solve using iteration.
5. All problems can be solved using iteration.

Concept of Local, Global and Static Variables:

Local Variables*(automatic or internal variable):*

The automatic variables are always declared within a function or block and are local to the particular function or block in which they are declared. As local variables are defined within the body of the function or the block, other functions can not access these variables. The compiler shows errors in case other functions try to access the variables.

Default **Initial value** of such type of variable is an unpredictable value which is often called garbage value. The **scope** of it is local to the block in which the variable is defined. Again, its **life** is till the control remains within the block in which the variable is defined. The keyword auto may be used while declaration of variable. For example, auto int a; But this is normally not done.

/*An example to illustrate local variable*/

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

```
long int fact(int n)
    {
        int i;
        long int f=1;
        for(i=1;i<=n;i++)
              f*=1;
       return(f);
        }

void main()
      {
        int num=5;
        printf("The factorial of 5 is%ld",fact(num));
        }
```

Here, the variables n, i , and f are local to function fact() and are unknown to main() function.

Global Variables*(External):*

The external or global variables are those variables declared outside any block or function. The default initial value for these variables is zero. The scope is global i.e. within the program. The life time is as long as the program's execution doesn't come to an end. The keyword extern can be used to identify the global variable. But, since the variables declared outside any block are global by default, there is no need of using extern.

An example to illustrate the global variables,

```
 #include<stdio.h>
#include<conio.h>
int a=10;
void fun()
      {
  a=20;
printf("\t%d",a++);
 }
```

```
void main()
      {
       printf("\t%d",a);
       fun();
       printf("\t%d",a);
}
```

Output:

```
 10          20       21
```

Static Variables:

The static variables are defined using keyword **static.** The default initial value for this type of variable is zero if user doesn't initialize its value. Its scope is local to the block in which the variable is defined. Again, the life time is global i.e. its value persists between different function calls.

Examples:

With auto variables

```
increment()
      {
       int i=1;
       printf("%d\n",i);
       i++;
       }

void main()
{
        increment();
        increment();
        increment();
increment();
```

With static variables

```
increment()
      {
          static int i=1; /* here  i is static variable*/
           printf("%d\n",i);
           i++;
        }

  void main()
 {
          increment();
          increment();
           increment();
   increment();
```

```
    }                                                    }
```

Output :

```
    1       1
    1       2
    1       3
    1       4
```

The static variable doesn't disappear when the function is no longer active. Their values persist. if control comes back to the same function again, the static variables have the same values they had last time around.

**********END UNIT #8**********

Unit Nine

Pointers

A pointer is a variable that stores a memory address of a variable. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variables but is always preceded by ' *' (asterisk) operator.

Applications of a pointer:

There are a number of reasons for using pointers. Some of them are:

1) Pointers can be used to pass information back and forth between a function and its reference point.
2) Pointers provide a way to return multiple data items from a function via function arguments.
3) Pointers provide an alternative way to access individual array elements.
4) They increase the execution speed as they refer address.

Pointer declaration

Pointer variables can be declared as follows:

Syntax:

data-type * variable_name;

*E*xamples:

```
int *x; float *f; char *y;
```

In the first statement,'x' is an integer pointer and it tells to the compiler that it holds the address of any integer variable. In the same way 'f' is a float pointer which stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.

Here * is called pointer or indirection operator. Normal variable provides direct access to their own values whereas a pointer provides indirect access to the values of the variable whose address it stores.The indirection operator(*) is used in two distinct ways with pointers, declaration and dereference.When the pointer is declared, the star indicates that it is a pointer, not a normal variable.When the pointer is dereferenced, the indirection operator indicates that the value at that memory location stored in the pointer is to be accessed rather than the address itself.Also note that * is the same operator that can be used as the multiplication operator. The compiler knows which operator to call, based on the context. An example

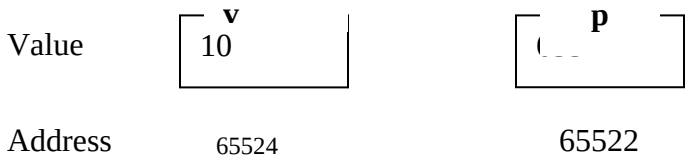
```
void main( )
{
 int v=10,*p; // Here, * indicates p as pointer
```

```
    p=&v;
    printf("\n address of v=%u",p);
    printf("\n value of v=%d",*p); // Here, * indicates value at address pointed by p
    printf("\n address of p=%u",&p);
}
```

Output:

address of v=65524
value of v=10
address of p=65522

It can be illustrated as



Here v is an integer variable and its value is 10. The variable p is declared as pointer variable. The statement p=&v assigns address of 'v' to 'p' i.e. 'p' is the pointer to variable 'v'. To access the address and value of 'v' pointer 'p' can be used. The value of 'p' is nothing but address of the variable 'v'. To display the value stored at that location *p is used. The address of the pointer variable can be accessed using '&' operator. Pointer variable can also be initialized at the time of declaration. It is done as follw.

```
int a;
int *p=&a;
```

Relationship between 1-Dimensional array and pointer

Array name by itself is an address or pointer. It points to the address of the first element (0th element of an array). Thus, if x is a one dimensional array, then address of the first array element can be expressed as either &x[0] or simple as x. The address of the second array element can be written as either &x[1] or as x+1 and so on. In general, the address of the array element i+1 can be expressed as either &x[i] or as x+i. In the expression x+i, x represents array name (address of first element) whose elements may be integers, characters, float etc and i

represents integer quantity. Thus, here x+i specifies an address that is a certain number of memory cells beyond the address of the first array element. Again, x[i] and *(x+i) both represent the content of that address.

Example**:**

To display array element with their address using array name as a pointer.

```
void main()
{
    int x[5]={2,4,6,8,10}, k;
    printf("\n element no. element address");
    for(k=0;k<5;k++)
    printf("\n x[%d]=\t%d %u",k,*(x+k), x+k);

    getch();
    }
```

output

Element no.	Element	address
X[0]	2	4056
X[1]	4	4058
X[2]	6	4060
X[3]	8	4062
X[4]	10	4064

In the above program element k acts as the element number and its value varies from 0 to 4. When it is added with an array name 'x' i.e. with address of the first element, it points to the consecutive memory location. Thus, element number, element and their addresses are displayed.

Another example:

Write a program to calculate average percentage of 10 students in a class using pointer. Read percentage of individual student from user.

Solution:

```
void main()
```

```
{
int per[10],i,sum=0;
clrscr();
printf("Enter percentage of each student\n");
for(i=0;i<10;i++)
 {
   scanf("%d",per+i);
   sum+=*(per+i);
 }
printf("\nThe average is=%d",sum/10);
 getch();
}
```

Pointers and 2-Dimensional arrays

Multi-dimensional array can also be represented with an equivalent pointer notation as in single dimensional array. A two dimensional array is actually a collection of one dimensional arrays. Therefore we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. Its general syntax is as

Data_type (*ptr_variable)[expression2];

Instead of

Data_type array[expression1][expression2];

Example:

Suppose x is a two dimensional integer array having 4 rows and 5 columns. We can declare x as

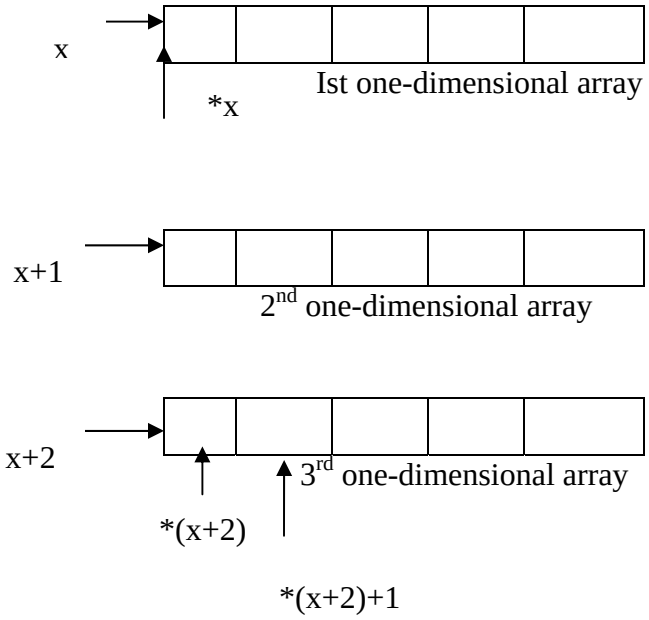
int (*x)[5];

rather than

int x[4][5];

Here, in first declaration, x is defined to be a pointer to a group of contiguous one dimensional 5-element integer arrays. The x points to the first 5-element array, which is actually

first row of the two dimensional array. Similarly, x+1 points to the second 5-element array, which is the second row of the two dimensional array. It is illustrated as



An example:

```
 void main()
    {
  int p[2][3]={   {1,2,3},
                   {4,5,6}
                  };
   clrscr();
   printf("p=%u\tp+1=%u",p,p+1);
printf("\n*p=%u\t*(p+1)=%u",*p,*(p+1));
   printf("\n*(p+0)+1=%u\t*(p+1)+1=%u",*(p+0)+1,*(p+1)+1);
   printf("\n*(*(p+0)+1)=%u\t*(*(p+1)+1)=%u",*(*(p+0)+1),*(*(p+1)+1));
getch();
    }
```

Output:

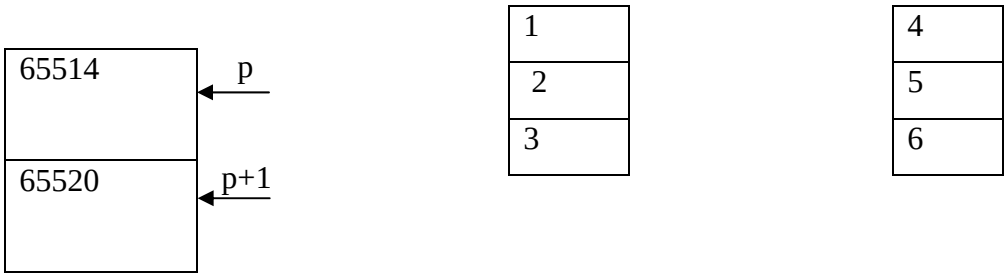
p=65514 p+1=65520

*p=65514 *(p+1)=65520

*(p+0)+1=65516 *(p+1)+1=65522

((p+0)+1)=2 *(*(p+1)+1)=5

It can be illustrated as



Example:

Write a program to add two 2*3 matrixes using pointer.

```
void main()
 {
    int (*a)[3],(*b)[3],*(sum)[3],i,j;
    clrscr();
    printf("Enter first matrix:\n");
    for(i=0;i<2;i++)
          for(j=0;j<3;j++)
            scanf("%d",*(a+i)+j);

printf("Enter second Matrix:\n");
for(i=0;i<2;i++)
```

```
        for(j=0;j<3;j++)
          scanf("%d",*(b+i)+j);

  printf("\nThe sum matrix is:\n");
   for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
         {
          *(*(sum+i)+j)= *(*(a+i)+j)+ *(*(b+i)+j);
          printf("\t%d",*(*(sum+i)+j));

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

Pointer Operations

1) Address of any variable can be assigned to a pointer variable. E.g.

   p=&a; q=&b ;

   where p & q are pointer variables and a & b are simple variables.

2) Content of one Pointer can be assigned to other pointer provided they point to same data type. E.g.

   int *p, *q;

   p=q;

3) Integer data can added to or subtracted from pointer variables.

   E.g. int *p;

   p+1;

4) One pointer can be subtracted from other pointer provided they point to elements of same array. E.g.

   int a[3],*pf,*pl;

```
        pf=a;
        pl=a+2;
    printf(“%p”,pl-pf);
```

5) There is no sense in assigning an integer to a pointer variable.
6) Pointer variable can not be multiplied and added.
7) Null value can be assigned to a pointer variable.

Passing pointer to a function

Refer to note and example of **call by reference** in Chapter 7 i.e. Function

An example:

Write a program to covert upper case letter into lower and vice versa using call by reference.

```
void conversion(char *); //function prototypes
void main()
 {
  char input;
  clrscr();
  printf("Enter Character of Ur choice\n");
  scanf("%c",&input);
  conversion(&input);
  printf("\nThe corresponding character is\t%c",input);
  getch();
}

void conversion(char *c)
  {
  if(*c>=97 && *c<=122)
     *c=*c-32;
  else if(*c>=65 && *c<=90)
    *c=*c+32;
  }
```

Dynamic Memory Allocation:

The process of allocating and freeing memory at run time is known as Dynamic Memory Allocation. This conserves the memory required by the program and returns this valuable resource to the system once the use of reserved space is utilized.

Since an array name is actually a pointer to the first element within the array, it is possible to define the array as a pointer variable rather than as a conventional array. While defining conventional array, system reserves fixed block of memory at the beginning of program execution which is inefficient but this does not occur if the array is represented in terms of a pointer variable. The use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as DMA.

There are four library functions malloc(), calloc(), free() and realloc() for memory management. These functions are defined within header file stdlib.h and alloc.h .They are described as follow.

i) malloc()

It allocates requested size of bytes and returns a pointer to the first byte of the allocated space. Its syntax is as

ptr=(data_type*) malloc(size_of_block);

Here, ptr is a pointer of type data_type. The malloc() returns a pointer to an area of memory with size size_of_block.

An example:

```
x=(int*) malloc(100*sizeof(int));
```

A memory space equivalent to “100 times the size of a integer (i.e. 100*2bytes =200 bytes ” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type int.

ii) calloc()

It allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory. The function calloc() allocates multiple blocks of storage, each of the same size

and then sets all bytes to zero. Thus, it is normally used for requesting memory space at run time for storing derived data types such as arrays and structure. Its syntax is

ptr=(data_type*) calloc(no_of_blocks, size_of_each_block);

An example:

```
x=(int*) calloc(5, 10*sizeof(int));
```

The above statement allocates contiguous space for 5 blocks, each of size 20 bytes i.e. we can store 5 arrays, each of 10 elements of integer types.

iii) free();

This built-in function frees previously allocated space. The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be done using free() function. Thus, this function is used to release the space when it is not required. Its syntax is

free(ptr);

Here, ptr is a pointer to a memory block which has already been created by malloc() or calloc() function.

iv) realloc();

This function is used to modify the size of previously allocated space. Sometimes, the previously allocated memory is not sufficient and we need additional space and sometime the allocated memory is much larger than necessary. In both situations, we can change the memory size already allocated with the help of function realloc(). Its syntax is as

If the original allocation id done by the statement

```
ptr=malloc(size);
```

then, reallocation of space may be done by the statement

ptr=realloc(ptr,newsize);

This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block.

An example related to realloc()

```
  #include<stdlib.h>
void main()
 {
 char *name;
 name=(char*)malloc(10);
strcpy(name,"Manahari");
 printf("\n Name=%s",name);
 name=(char*)realloc(name,20);
 strcpy(name,"Manahari Sijapati");
 printf("\n Name=%s",name);
 getch();
 }
```

Another Example:

Write a program to read number of students and their marks and display the average of entered marks. Use array as pointer instead of conventional array to represent marks of different students.

```
#include<stdlib.h>
    void main( )
         {
          int s,i;
          float *p,sum=0,avg;
          clrscr();
          printf("\nHow many students are there?\n");
          scanf("%d",&s);
          printf("\n Enter marks of each students\n");
          p=(float*)malloc(s*sizeof(float));
            for(i=0;i<s;i++)
              {
                scanf("%f",p+i);
```

```
        sum+=*(p+i);
       }
     avg=sum/s;
    printf("\n The average marks is\t%f",avg);
    free(p);
    getch();
}
```

**********END UNIT #9**********

Unit Ten

Structure and Union

A structure is a collection of variables usually of different types that can be used as a single item to handle data more efficiently. An array can be used only to represent a group of data items that belong to the same type. But, if we want to represent a collection of data items of different types using a single name, array can not be used. At that situation, structure is used.

Defining a Structure:

Syntax:

```
struct   structure_name
         {
          type1 member_variable1;
          type2 member_variable2;
          .
          .
          typen member_variablen;
         };
```

Once structure_name is declared as new data type, then variables can be declared as

struct structure_name structure_variable;

An example:

Let us create a structure named student that has name, roll, marks and remarks as members.

```
struct student
         {
           char name[20];
           int roll;
           float marks;
           char remarks;
         };
```

Here, student is structure name and its members are name, roll, marks and remarks.Then different variables of type **struct student** can be declared as

struct student st;

Here, struct student is new data type and st is variable of that type. The multiple variables are declared as

struct student st1, st2, st2;

Each variable has its own copy of member variables. The member variables are accessed using dot (.) operator. For example, st1.name is member variable, name, of st1 structure variable and st2.name is the member variable, name, of second structure variable st2. Similarly, st1.roll is the

member, roll, of first structure variable st1 and st2.roll is the member, roll, of second structure variable st2.

The structure definition and variable declaration can be combined as following

```
struct student
        {
          char name[20];
          int roll;
          float marks;
          char remarks;
        }st1,st2;
or

struct
        {
          char name[20];
          int roll;
          float marks;
          char remarks;
        }st1,st2;
```

How structure elements are stored?

The elements of a structure are always stored in contiguous memory locations. This can be illustrated as

```
void main()
{
  struct student
   {
   int roll;
   float marks;
```

```
    char remarks;
  };
  struct student st={200,60.5,'P'};

  printf("\nAddress of roll=%u",&st.roll);
  printf("\nAddress of marks=%u",&st.marks);
  printf("\nAddress of remarks=%u",&st.remarks);
  getch();
}
```

Output:

Address of roll=65518

Address of marks=65520

Address of remarks=65524

st.roll	st.marks	st.remarks
200	60.5	'P'
65518	65520	65524

Processing a Structure:

The members of a structure are usually processed individually, as separate entity. A structure member can be accessed by writing

structure_variable.member

Here, structure_variable refers to the name of a structure-type variable and member refers to the name of a member within the structure. Again, dot that separates the variable name from the member name.

In the case of nested structure (if a structure member is itself a structure), the member within inner structure is accessed as

structure_variable.member.submember

This will be clear in nested structure section.

An Example:

Create a structure named student that has name, roll, marks and remarks as members. Assume appropriate types and size of member. Write a program using structure to read and display the data entered by the user.

```
void main()
  {
    struct student{              {
                    char name[20];
                    int roll;
                    float marks;
                    char remark;
                    };
   struct student s;
   printf("Enter name:\t");
  gets(s.name);
  printf("\nEnter roll:\t");
  scanf("%d",&s.roll);
  printf("\n Enter marks\t");
  scanf("%f",&s.marks);
  printf("Enter remark p for pass or f for fail\t");
  s.remark=getche();
  printf("\n\n\nThe Detail Information is\n");
printf("\Name=%s\tRoll=%d\nMarks=%f\tRemark=%c",s.name,s.roll,s.marks,s.remark);
  getch();
  }
```

Array of structure:

Like array of int, float or char type, there may be array of structure. In our previous structure example, if we want to keep record of more students, we can make more structure variables like st1, st2, st3, ….so on. But this technique is inefficient. At this situation, we can use array of structure. The array of structure is declared as

struct structure_variable[size_of_array];

This is illustrated by following program.

Example:

Modify the above program such that it works to keep records of five students. Use array of structure.

```
void main()
  {
  struct student {
char name[20];
int roll;
float marks;
char remarks;
};
  struct student s[5];  int i; //array declaration
 for(i=0;i<5;i++)
 {
   printf("\n\n\nEnter information of Student [%d]\n",i);
   printf("Enter name:\t");
   scanf("%s",&s[i].name);
   printf("\nEnter roll:\t");
   scanf("%d",&s[i].roll);
   printf("\n Enter marks\t");
   scanf("%f",&s[i].marks);
   printf("Enter remarks p for pass or f for fail\t");
   s[i].remarks=getche();
   }
```

```
   printf("\n\nYour Entered Information is:\n");
  for(i=0;i<5;i++)
  {
printf("\nStudent %d\n",i);
printf("\Name=%s\tRoll=%d\nMarks=%d\tRemarks=%c",s[i].name,s[i].roll,s[i].marks,s[i].remar
ks);
    }
   getch();
  }
```

Structure within another structure (Nested Structure):

One structure can be nested within another structure in C. Let us consider an structure mark which has members subject and marks. This can be nested within another structure student. This can be done as follow.

```
struct mark{
        char subject[20];
        float marks;
         };
```

This structure can be nested within another structure as its member.

```
struct student {
        char name[20];
        int roll;
        struct mark studentM;
        char remark;
        }s;
```

Here, structure student contains a member **studentM**which is itself a structure with two members. The members within structure mark are accessed as

s.studentM.subject and s.studentM.marks

But the members within student structure are accessed as usual

s.name, s.roll and s.remark

The above nested structure can be nested in alternative way.

```
struct student {
        char name[20];
        int roll;
char remark;
        struct mark{
          char subject[20];
          float marks;
           }studentM;
 }s;
```

Example:

Create a structure named mark that has subject and marks as its members. Include this structure as a member for student structure created in program1. Use this structure to read and display student's name, roll, subject, marks and remarks.

```
void main()
 {
 struct mark{
          char subject[20];
          float marks;
           };
 struct student {
        char name[20];
        int roll;
        struct mark studentM;
        char remark;
         };
  struct student s;
    printf("Enter name:\t");
    scanf("%s",s.name);
    printf("\nEnter roll:\t");
```

```
    scanf("%d",&s.roll);
    printf("\nEnter Subject Name:\t");
    scanf("%s",s.studentM.subject);
    printf("\n Enter marks\t");
   scanf("%f",&s.studentM.marks);
   printf("Enter remark p for pass or f for fail\t");
   s.remark=getche();
  printf("\n\n\nThe Detail Information is\n");
printf("\Name=%s\tRoll=%d\nSubject=%s\tMarks=%f\nRemark=%c",s.name,
                        s.roll,s.studentM.subject,s.studentM.marks,s.remark);
 getch();
 }
```

Structure and Pointer

To store address of a structure type variable, we can define a structure type pointer variable as normal way. Let us consider an structure book that has members name, page and price. It can be declared as

```
struct book
    {
      char name[10];
     int page;
     float price;
      };
```

Then, we can define structure variable and pointer variable of structure type. This can be done as

```
struct book b, *bptr;
```

Here,

b is the structure variable and bptr is pointer variable which points address of structure variable. It can be done as

```
bptr=&b;
```

Here, the base address of b can assigned to bptr pointer.

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

ptr_variable->member

Here, -> is called arrow operator and there must be pointer to the structure on the left side of this operator.

Example:

Create a structure named book which has name, pages and price as member variables. Read name of book, its page number and price. Finally display these members' value. Use pointer to structure instead of structure itself to access member variables.

```
 void main()
{
   struct book
        {
        char name[20];
        int pages;
        float price;
         };
 struct book b={"Let us C",230,456.50},*ptr; // structure initialization
 ptr=&b;
 clrscr();
 printf("\nName=%s\tPages=%d\tPrice=%f",ptr->name,ptr->pages,ptr->price);
                  /* same as b.name, b.pages and b.price */
 getch();
}
```

Here, ptr->name refers the content at address pointed by ptr variable.

Passing structure to a Function:

Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure. Let us take an example in which there is structure employee that has name, sex and salary as members. These members can be passed to function to display their value.

Example:

In this example, member variables are displayed by passing individual elements to a function display().

```
void display(char name[],char s, int sal)
 {
  printf("\nName=%s\tSex=%c\tSalary=%d",name,s,sal);
 }

void main()
 {
 struct employee
    {
   char name[20];
   char sex;
   int salary;
  };
 struct employee e={"abc",'M',10000};
 display(e.name,e.sex,e.salary);
                          /*function call in which each elements are passed individually*/
 getch();


 }
```

Another Example:

Let us consider above structure but pass whole structure variable to function.

```
struct employee
  {
   char name[20];
   char sex;
   int salary;
  };   // structure is declared outside main()

void display(struct employee emp)
 {
  printf("\nName=%s\tSex=%c\tSalary=%d",emp.name,emp.sex,emp.salary);
 }

void main()
 {
 struct employee e={"abc",'M',10000};
 clrscr();
 display(e);  /* function call*/
 getch();
 }
```

In above example, the whole structure variable e is passed to a function display(). In formal argument of function display(), we need structure variable to receive structure variable sent by main() function. Thus, we should declare structure outside the main() function such that it is recognized also by function display().

Union:

Unions are similar to structure. Its syntax and use is similar to structure. It also contains members whose individual data types may differ from one another. The distinction is that all members within union share the same storage area of computer memory, whereas each member within a structure is assigned its own unique storage. Thus, unions are used to conserve memory. Since same memory is shared by all members, one variable can reside into memory at a time. When

another variable is set into memory, the previous is replaced i.e. previous can not persist. Thus, unions are useful for applications involving multiple members where values need to be assigned to all of the members at any one time. Therefore, although a union may contain many members of different types, it can handle only one member at a time.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. For example

The union is created as

```
union student
        {
          int roll;
          float marks;
        } ;
```

Here, the union student has members roll and marks. The data type of roll is integer which contains 2 bytes in memory and the data type of marks is float which contains 4 bytes in memory. As all union members share same memory, the compiler allocates larger memory (i.e. 4 bytes in this case).

The declaration of union and its variable is similar to that of structure. The union variable can be passed to function and it can be member of any other structure. The union can be illustrated as following examples;

Example:

Create an union named student that has roll and marks as member. Assign some values to these members one at a time and display the result one at a time.

```
void main()
 {
  union student
        {   int roll;
            float marks;
         };
union student st;
 st.roll=455;
```

```
 printf("\nRoll=%d",st.roll);
 st.marks=78;
 printf("\nMarks=\t%f",st.marks);
  }
```

output:

Roll=455

Marks= 78.000000

If two members are used simultaneously, the output is unexpected as following

```
void main()
 {
  union student
      {     int roll;
           float marks;
       };
 union student st;
  st.roll=455;
  st.marks=78;
printf("\nRoll=%d",st.roll);
printf("\nMarks=\t%f",st.marks);
  }
```

output:

Roll=0

Marks= 78.000000

Unit Eleven

Data Files

The input output functions like printf(), scanf(), getchar(), putchar(), gets(), puts() are known as console oriented I/O functions which always use keyboard and computer screen as the target place. While using these library functions, the entire data is lost when either the program is terminated or the computer is turned off. Again, it becomes cumbersome and time consuming to

handle large volume of data through keyboard. These problems invite concept of data files in which data can be stored on the disks and read whenever necessary, without destroying data.

A file is a place on the disk where a group of related data is stored. The data file allows us to store information permanently and to access and alter that information whenever necessary. Programming language C has various library functions for creating and processing data files. Mainly, there are two types of data files, one is **stream oriented** or **standard**or **high level** and another is **system oriented** or **low level** data files. In standard data files, the available library functions do their own buffer management whereas the programmer should do it explicitly in the case of system oriented files. Thus, system oriented data files are generally easier to work with and are therefore more commonly used.

The standard data files are again subdivided into text files and binary files. The text files consist of consecutive characters and these characters can be interpreted as individual data item. The binary files organize data into blocks containing contiguous bytes of information.

Opening and Closing a File:

While working with stream-oriented file, we need buffer area where information is stored temporarily in the course of transferring data between computer memory and data file. The buffer area is established as

FILE * ptr_varibale;

Where FILE is a special structure type that establishes the buffer area and ptr_variable is a pointer variable that indicates the beginning address of the buffer area. The special structure FILE is defined within header file stdio.h.

A data file must be opened before it can be created or processed. This is done by following syntax

ptr_variable=fopen(file_name, file_mode);

This associate file name with the buffer area and specifies how the data file will be utilized (File opening mode). The function fopen() returns a pointer to the beginning of the buffer area associated with the file. A NULL value is returned if the file can be opened due to some reason. After opening a file, we can process data files as requirement. Finally, data file must be closed at the program. This is done using other library function fclose() as below

fclose(ptr_variable);

File opening modes:

The file opening mode specifies that how the data file is utilized while opening i.e. as read only or write only or read/write file. There are mainly six modes. They are illustrated as followings:

i. **"r" (i.e. read):**

This opens an existing file for reading only. It file searches specified file. If file exists, it loads into memory and sets up a pointer which points to the first character in it. If file does not exist, it returns NULL. The possible operation is only- **reading from the file**

ii. **"w" (i.e. write):**

This mode of file opens a file for writing only. It searches specified file. If file already exist, the contents are overwritten. If file does not exist, a new file is created. It returns NULL, if it is unable to open the file. The possible operation is only- **writing to the file.**

iii. **"a" (i.e. append):**

It opens an existing file for appending (i.e. adding new information at the end of file). It searches specified file. If specified file exists, loads into memory and set up a pointer that points to the last character in it. If the file does not exist, a new file is created. It returns NULL if unable to open the file. The possible operation is –**adding new contents at the end of file.**

iv. **"r+" (i.e. read + write):**

It opens existing file for reading and writing. It searches the specified file. If the file exists, loads it into memory and set up a pointer which points to the first character in it. Again, it returns NULL if file doesn't exist. The possible operations are- **reading existing contents, writing new contents, modifying existing contents of the file.**

v. **"w+" (i.e. write + read):**

It opens file for reading and writing. If the specified file exists, its contents are destroyed. If file doesn't exist, the new file is created. It return NULL if unable to open file. The

possible operations are- **writing new contents, reading them back and modifying existing contents of the file.**

vi. **"a+" (i.e. append+ read):**

It opens an existing file for both reading and appending. A new file will be created if specified file doesn't exist. The possible operations are- **reading existing contents, appending new contents to end of file but it can not modify existing content.**

The library functions related to reading and writing to a file

i. **fgetc()**

It is used to read a character from a file. Its syntax is

char_variable=fgetc(file_ptr_variable);

ii. **fputc()**

It is used to write a character to a file. Its syntax is

fputc(character or char_variable, file_ptr_variable);

iii. **fgets()**

It is used to read string from file. Its syntax is

fgets(string_variable, int_value or int_variable, file_ptr_variable);

Here int_value or int_variable denotes the number of character in a string.

iv. **fputs()**

It is used to write a string to a file. Its syntax is

fputs(string_variable, file_ptr_variable);

v. **fprintf()**

This function is formatted output function which is used to write some integer or float or char or string to a file. Its syntax is

fprintf(file_ptr_variable, "control_string", list_variables);

vi. fscanf()

This function is formatted input function which is used to read some integer or float or char or string from a file. Its syntax is

fscanf(file_ptr_variable, “control_string”,&list_variables);

Example 1:

Create a file named test.txt and write some text “Wel come to Acme” at the file.

```
void main()
 {
 FILE *fptr;
 clrscr();
 fptr=fopen("test.txt","w");
 if(fptr==NULL)
   {
   printf("\nFile can not be created");
   exit();
   }
 fputs("Wel come to Acme",fptr);
 fclose(fptr);
  }
```

Example 2:

Write a program to open the file created in example 1, read its content and display.

```
void main()
 {
 FILE *fptr;
  char s[100];
   clrscr();
 fptr=fopen("test.txt","r");
```

```
 if(fptr==NULL)
    {
    printf("\nFile can not be opened");
    exit();
    }
 fgets(s,100,fptr);
 printf("\nFrom File\n%s",s);
 fclose(fptr);
getch();
 }
```

Example 3:

Create a program which asks file name from user and create it to write certain lines of text. It should write until the user hits enter at the beginning of line.

```
void main()
 {
 FILE *fptr;
 char s[200],fileName[20];
 clrscr();
 printf("\nEnter file name\n");
 gets(fileName);
 fptr=fopen(fileName,"w");
 if(fptr==NULL)
    {
    printf("\nFile can not be created");
    exit();
    }
  while(strlen(gets(s))!=0)
```

```
  {
   fputs(s,fptr);
  fputs("\n",fptr);
   }
fclose(fptr);
 }
```

Example 4:

Write a program to create file named test.txt and write some text to it. Write texts character by character using fputc() function. It should write until user hits enter key. Again, read its contents and display.

```
void main()
 {
 FILE *fptr;
 char c;
 clrscr();
 fptr=fopen("test.txt","w+");
 if(fptr==NULL)
   {
   printf("\nFile can not be created");
          }
  printf("Enter Text\n");
 while((c=getchar())!='\n')
    {
    fputc(c,fptr);
    }
  rewind(fptr);  /*set file pointer at initial position*/
  printf("\nThe Text From File\n");
 while((c=fgetc(fptr))!=EOF)
  {
```

```
    putchar(c);
    }
  fclose(fptr);
 getch();
 }
```

Example 5:

Write a program to append some text to a certain file. Take file name from user.

```
void main()
 {
 FILE *fptr;
 char s[100],fileName[20];
 clrscr();
 printf("\nEnter file name at which u want to append\n");
 gets(fileName);
 fptr=fopen(fileName,"a");
 if(fptr==NULL)
   {
   printf("\nFile can not be created or opened");
           }
  printf("Enter Text to be appended\n");
  while((c=getchar())!='\n')
     {
     fputc(c,fptr);
     }
   fclose(fptr);
   getch();
 }
```

Example 6

Create a program to create a data file and writes the natural number 1 to 20 and then read the numbers from file to display as twice of the stored numbers.

```
void main()
 {
 FILE *fptr;
 int i,numFromFile;
 clrscr();
 fptr=fopen("test1.txt","w");
 if(fptr==NULL)
   {
   printf("\nFile can not be created");
   exit();
   }
 for(i=1;i<=20;i++)
   {
   fprintf(fptr,"\n%d",i);
   }
  fclose(fptr);
 fptr=fopen("test1.txt","r");
 printf("\nThe output from File\n");
for(i=1;i<=20;i++)
  {
  fscanf(fptr,"%d",&numFromFile);
  numFromFile=2*numFromFile;
  printf("\t%d",numFromFile);
  }

 fclose(fptr);
 getch();
 }
```

Example 7:

Write a program that opens a file and copies all its content to another file. Take source and destination file from user.

```
void main()
 {
 FILE *fptrSource,*fptrDest;
 char ch,sourceF[20],destF[20];

 clrscr();
 printf("\nEnter Source File name\t");
 gets(sourceF);
 printf("\nEnter Destination File\n");
 gets(destF);
 fptrSource=fopen(sourceF,"r");
 if(fptrSource==NULL)
   {
   printf("\nFile can not be opened");
   exit();
   }
 fptrDest=fopen(destF,"w");
 if(fptrDest==NULL)
  {
  printf("\nDestination File can not be created\n");
  exit();
  }
  while((ch=fgetc(fptrSource))!=EOF)
     {
     fputc(ch,fptrDest);
     }
 fclose(fptrSource);
 fclose(fptrDest);
```

```
getch();
 }
```

End **O**f **F**ile (**EOF):**

The EOF is a special character, whose ASCII value is 26, which indicates the end of a file in text mode of file. While writing data to a file, this character is inserted automatically after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function would return the EOF signal to the program indicating the last point of file. An attempt to read after EOF might either cause the program to terminate with an error or result in an infinite loop situation. Thus, the last point of file is detected using EOF while reading data from file. Without this mark, we can not detect last character at the file such that it is difficult to find up to what time the character is to be read while reading data from the file.

Binary Mode of Data Files:

In Binary mode of data file, the opening mode of text mode file is appended by a character 'b' i.e.

"r" is replace by "rb"

"w" is replace by "wb"

"a" is replace by "ab"

"r+" is replace by "r+b"

"w+" is replace by "w+b"

"a+" is replaced by "a+b"

The default mode is text mode. Thus when simple "r" is written as mode of opening, this is assumed as text mode. We can write "rt" in place of "r" and so on in the case of text of file. The syntax for all above mode in the case of binary mode is similar to the previous one i.e. text mode like this

```
fptr=fopen("data.txt","rb");
```

Difference between Binary mode and Text Mode

There are mainly three difference between binary and text mode. They are as follow.

i. In text mode, a special character EOF whose ASCII value is 26 is inserted after the last character in the file to mark the end of file. But, there is no such special character present in the binary mode. The binary mode files keep track of the end of file from the number of characters present in the directory entry of the file.

ii. In text mode of file, text and numbers are stored as one character per byte. For example, the number 1234 occupies two bytes in memory but it occupies 4 bytes (one byte per character) in the file. But, in binary mode, the number occupies the number of bytes as it occupies in the memory. Thus, the above number occupies two bytes in file also in the case of binary mode.

iii. In text mode, a new line character is converted into the carriage return-linefeed combination before being written to the disk. Likewise, the carriage return-line feed combination on the disk is converted into a new line when the file is read by a C program. However, these conversions doesn't take place in the case of binary mode.

An example for binary mode of file:

Write a C program to write some text "Welcome to AcmeCollege" to a data file in binary mode. Read its content and display it.

```
void main()
 {
 FILE *fptr;
 char c;
 clrscr();
 fptr=fopen("test.txt","w+b");
 if(fptr==NULL)
   {
   printf("\nFile can not be created");
   }
  fputs("Welcome to AcmeCollege",fptr);
  rewind(fptr);
```

```
  printf("\From File\n");
  while(!feof(fptr))
    {
    printf("%c",getc(fptr));
    }
  fclose(fptr);
 getch();
 }
```

Example of writing structure elements to a file,

Write a C program to create a structure named book having elements name, page and price. Enter the members from keyboard. Write these data to a file.

1) Using fprintf() function

```
void main()
{
FILE *fp;
struct book
  {
  char name[20];
  int page;
  float price;
  };
struct book b;
fp=open("stTest.txt","wb");
if(fp==NULL)
  {
  printf("\nError");
  exit();
  }
printf("\nEnter name of book\n");
gets(b.name);
```

```
printf("\nEnter number of pages\n");
scanf("%d",&b.page);
printf("\nEnter price of book\n");
b.price=89;
fprintf(fp,"%s%d%f",b.name,b.page,b.price);
printf("\nThe file has been createed\n");
fclose(fp);
}
```

2. Using fwrite() function:

```
void main()
   {
  ….
 …….
 …….
fwrite(&b, sizeof(b),1,fp); /* 1 means one structure*/
fclose(fp);

    }
```

Syntax for fwrite() and fread()

fwrite(&str_variable, sizeof(str_variable), number_of_structure, file_ptr);

fread(&str_variable, sizeof(str_variable), number_of_structure, file_ptr);

Unit Twelve

Graphics

Computer graphics is one of the most powerful and interesting aspects of computers. There is a lot that we can do in graphics apart from drawing figures of various shapes. All video games, animation, multimedia predominantly works with computer graphics. To work with graphics, C has various library functions. Some of built in functions which are defined within header file graphics.h are illustrated in this chapter.

Let us see one example which demonstrates graphics initialization and some built in functions related to graphics.

```
#include<graphics.h>
void main()
{
int gDriver=DETECT,gMode;
int polyArray[]={200,100,250,150,300,200,350,200,300,150,200,100};
initgraph(&gDriver,&gMode,"c:\\tc\\bgi");
setcolor(GREEN);
line(200,100,300,250); /*line from (200,100) to (300,250) */
circle(300,300,50); /*circle at center (300,300) having radius 50 */
ellipse(150,200,0,360,100,50);          /* ellipse at center (150,200) and angle from 0 to
                                          360 and the xradius 100, yradius 50*/
arc(200,300,300,90,80);  /* arc having center at (200,300) and radius 80 from starting
                            angle 300 to end angle 90 */
rectangle(50,100,350,400);  /* rectangle of diagonal points at (50,100) and (350,400)*/
drawpoly(6,polyArray); /* polygon of 6 corner points*/
getch();
closegraph();
}
```

Explanation:

The header file **graphics.h** should be included to use graphics related built in functions. Again, the variable **gDriver** is an integer variable that specifies graphics driver to be used.Device drivers are small programs which talk directly to the hardware. Graphics drivers are a subset of device drivers and are applicable only in the graphics mode. The initialization of this variable to DETECT is to set another variable gMode to the highest resolution available for the detected driver. The value of this variable gDriver may be another than DETECT (e.g. 1 for CGA, 9 for VGA.). The variable **gMode** is an integer that specifies initial graphics mode.

The built in function **initgraph()** is for graphics initialization. The syntax for this is as

initgraph(&graphics_driver, &graphics_mode, path_to_driver);

path_to_driver specifies the directory path where initgraph() looks for graphics drivers (.BGI files). This function initializes the graphics system by loading a graphics driver from disk then putting the system into graphics mode. Iinitgraph() also resets all graphics settings like color, current positions etc to their default values. Normally, initgraph()
loads a graphic driver by allocating memory for driver than loading the appropriate .BGI file from disk.

The another built in function **setcolor(GREEN)**sets the color GREEN as drawing color. The function **closegraph()** shutdown the graphics system. It de-allocates all memory allocated by the graphics system. It then restores the screen to the mode it was in before initgraph() function called. The syntax for other function is given as below

i. **line():**

line(x1,y1,x2,y2);

It draws line from point (x1,y1) to another point (x2,y2).

ii. **circle();**

circle(x,y,radius);

It draws a circle at center co-ordinate (x,y) and having radius radius.

iii. **ellipse();**

ellipse(x, y, staring_angle, end_angle, xradius, yradius) ;

It draws a ellipse having center at co-ordinate (x,y) and radii xradius & yradius. The drawing angles are starting_angle and end_angle. Generally, stating_angle is 0 and end_ange is 360 for a complete ellipse.

iv. **arc();**

arc(x, y, starting_angle, end_angle, radius);

It draws arc having center point at co-ordinate (x,y) and certain radius.

v. **rectangle();**

rectangle(x1,y1,x2,y2);

It draws a rectangle having diagonal points at co-ordinates (x1,y1) and (x2,y2).

vi. **drawpoly();**

drawpoly(number_of_poly_points, array_of_co-ordinates_of_corners);

It draws required polygon. The number_of_poly_points denotes number of points (i.e. for pentagon 5 , for hexagon 6 and so on). Again, array_of_co-ordinates_of_corners denotes an array in which the corner point's co-ordinates are stored.

The other various useful built in functions related to graphic are

i. getmaxx()
ii. getmaxy()
iii. outtextxy()
iv. putpixel()
v. restorecrtmode()
vi. moveto()
vii. lineto()
viii. moverel()
ix. linerel()

Some Graphics Mode Graphics Functions:

putpixel( ): Plot a point with specified color. Its syntax is

Putpixel (int x, int y, int color),

getpixel( ): gets color of specified pixel. Its syntax is

integer_variable = getpixel (int x, int y);

setcolor( ): It changes current drawing/ foreground color. Its syntax is

setcolor (int color);

setbkcolor( ): It changes the background color. Its syntax is

setbkcolor(int color);

line( ): The line ( ) function can draw a line. The syntax of line( ) function is:

line(x1, y1, x2, y2),

where x1, y1, x2, y2 are integer type and they represent the coordinate (x1, y1) and (x2,y2). The above command draws a line joining two points with coordinates (x1, y1) and (x2,y2).

lineral( ): the function lineral (x, y) draws a line joining the current cursor position and apoint at a distance of x in the horizontal and y in vertical direction.Note that x and y both are integer type.

lineto( ): The function lineto (x, y) draws a line joining the current cursor position and a pointwith coordinates x and y.Where x and y both are integer type.

moveto( ): The function moveto (x, y) moves the cursor position to a point with coordinates xand y.

moveral( ): The function moveral (x, y) moves the current cursor position a relative distanceof x in x-direction & a relative distance of y in y-direction.

rectangle( ): The function rectangle (x1, y1, x2, y2) draw rectangle where point (x1, y1) isleft top corner point of rectangle and point (x2, y2) is right bottom corner.

circle( ): The function circle (x, y, r) draws a circle of radius r. The coordinates of the centreof the circle is (x, y).

arc( ): Function arc (x, y, a1, a2, r) draws an arc on the screen starting from angle a1 to a2. The radius of the circle of which the arc forms a part is r and x, y are its centre coordinates.

For example:

arc (100, 45, 90, 30) ;

ellipse( ): The function ellipse (x1, y1, c1, c2, a1, a2, x, y) draws an ellipse of centre (c1, c2). The a1 and a2 are start and end angle of the arc x and y are the x-axis and y-axis radii.

drawpoly( ): Function drawpoly (int n, int p[ ]); draws n vertices of polygon. P is an array ofinteger numbers which gives x and y co-ordinates of the points to be joined. To draw a closedpolygon with n vertices, we must pass n+ 1 co-ordinate.

```
Int P={10, 75, 50, 25, 100, 25, 140, 75, 100, 125, 50, 125, 10, 75};
Drawpoly (7, P);
```

setlinestyle( ): This function can select different style of line. Its syntax is

setlinestyle (int style, patern, thickness)

The type of style and thickness is int type and the type of pattern is unsigned int type.Where style are SOLID_LINE, DOTTED_LINE, CENTRE_LINE, DASHED_LINE,USERBIT_LINE or integer number 0, 1, 2, 3, 4 respectively. The pattern is required only ifuser defined style (USERBIT_LINE) is used. We can specify it to zero. The thicknessparameter can have value NORM_WIDTH or THICK_WIDTH or integer value 1 or 3respectively.

fillpoly( ): It draws and fills polygon. Its syntax is

fillpoly (int n, int p[ ]);

|* Program to draw 3 concentric circle having radius 25m 50 and 75 units *|

```
#include <graphics.h>
main( )
{
int gd = DETECT, gm;
clrscr ( );
initgraph ( &gd, &gm, "C:\\tc\\bgi");
setcolor (3);
circle (150, 150, 25);
circle (150, 150, 50);
circle (150, 150, 75);
getch ( );
closegraphc ( );
}
```

|* Program to draw an arc *|

```
#include <graphics.h>
main ( )
{
int gd = DETECT, gm;
clrscr ( );
initgraph ( &gd, &gm, "C:\\tc\\bgi"),
arc (150, 150, 0, 90, 30);
getch ( );
closegraph ( );
}
```

|* Program to draw a polygon *|

```
# include <graphics.h>
main ( )
{
int gd = DETECT, gm;
int P [ ] = (10, 75, 50, 25, 100, 25, 140, 75, 100, 125, 50, 125, 10, 753);
clrscr ( );
initgraph ( &gd, &gm, "C:\\tc\\bgi");
drawpoly (7, P);
fillp[oly (7, P);
getch ( );
closegraph ( );
}
```

|* Program in c to draw a line, a circle, a rectangle and an ellipse *|

```
#include <graphics.h>
main( )
{
int gd, gm;
clrscr ( );
gd = DETECT;
initgraph ( &gd, &gm, "C:\\tc\\bgi")
setcolor (2);
line (150,150,250,250);
circle (300, 300, 25)
setcolor (5);
rectangle (0, 0, 100, 200);
ellipse (150, 250, 0, 360, 80, 60);
getch ( );
closegraph ( );
}
```

NOTE: Students are suggested to go through reference books and text books also.