<div style="border: 1px solid black">

Chapter – 1

</div>

Problem Solving using Computer

What is a problem?

Problem is defined as the difference between an existing situation and a desired situation, that is, in accordance with calculation; a problem is numerical situation and has complex form. Solution is desired situation and has simplest form. If a problem is solved by computing using machine called computer, then such process is called Problem Solving using Computer.

1.1 Problem Analysis

If you have studied a problem statement, then you must analyse the problem and determine how to solve it. First, you should know the type of problem that is, nature of problem. In programming point of view, the problem must be computing. At first you try to solve manually. If it is solvable manually by using your idea and knowledge, then you can use such idea and principle in programming and solve the problem by using computer. So, you must have well knowledge about a problem. In order to get exact solution, you must analyse the problem. To analyse means you should try to know the steps that lead you to have an exact solution.

Suppose you are asked by your father to solve an arithmetic problem and you are not familiar with the steps involved in solving that problem. In such a situation, you will not be able to solve the problem. The same principle applies to writing computer program also. A programmer cannot write the instruction to be followed by a computer unless the programmer knows how to solve the problem manually.

Suppose you know the steps to be followed for solving the given problem but while solving the problem you forget to apply some steps or you apply the calculation steps in the wrong sequences. Obviously, you will get a wrong answer. Similarly, while writing a computer program, if the programmer leaves out some of the instructions for the computer or writes the instructions in the wrong sequences, then the computer will calculate a wrong answer. Thus to produce an effective computer program, it is necessary that the programmers write each and every instruction in the proper sequence. However, the instruction sequence (logic) of a computer program can be very complex. Hence, in order to ensure that the program instructions are appropriate for the problem and are in correct sequence, program must be planned before they are written.

1.2   Algorithm Development & Flowcharting:

The term <u>algorithm</u> may be formally defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired result will be obtained. An algorithm must posses the following characteristics:

1. Each and every instruction should be precise and unambiguous.
2. Each instruction should be such that it can be performed in a finite time.
3. One or more instruction should not be repeated infinitely. This ensures that the algorithm will ultimately terminate.
4. After performing the instructions, that is after the algorithm terminates, the desire4d results must be obtained.

Problem:

There are 50 students in a class who appeared in their final examination. Their mark sheets have been given to you. Write an algorithm to calculate and print the total number of students who passed in first division.

Algorithm:

Step 1: Initialize Total First Division and Total Mark sheet checked to zero i.e.

total_first_div = 0;

total_marksheet_chkd = 0;

Step 2: Take the mark sheet of the next student.

Step 3: Check the division column of the mark sheet to see if it is I: if no, go to step 5.

Step 4: Add 1 to Total First Division i.e.

total_first_div +1;

Step 5: Add 1 to Total Mark sheets checked i.e.

total_marksheet_chkd +1;

Step 6: Is Total Mark sheets checked = 50: if no go to step 2

Step 7: Print Total First Division.

Step 8: Stop (End)

The above mentioned example is simpler one but development of an algorithm of a complex problem is very difficult. It may also be noted that in order to solve a given problem, each and every instruction must be strictly carried out in a particular sequence.

Flowchart:

A flowchart is a pictorial representation of an algorithm that uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed.

Computer Programming in C

Normally, an algorithm is first represented in the form of a flowchart and the flowchart is then expressed in some programming language to prepare a computer program. The main advantage of this two steps approach in program writing is that while drawing a flowchart one is not concerned with the details of the elements of programming language. Since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected more easily than in the case of a program. Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.

A flowchart, therefore, is a picture of the logic to be included in the computer program. It is simply a method of assisting the program to lay out, in a visual, two dimensional format, ideas on how to organize a sequence of steps necessary to solve a problem by a computer. It is basically the plan to be followed when a program is written. It acts like a road map for a programmer and guides him/her how to go from starting point to the final point while writing a computer program.

Experienced programmers sometimes write programs without drawing the flowchart. However, for a beginner it is recommended that a flowchart be drawn first in order to reduce the number of errors and omissions in the program. It is a good practice to have a flowchart along with a computer program because a flowchart is very helpful during the testing of the program as well as while incorporating further modifications in the program.

Flowchart Symbols:

A flowchart uses boxes of different shapes to denote different types of instructions. The communication of program logic through flowcharts is made easier through the use of symbols that have standardized meanings. For example, a diamond always means a decision. Only a few symbols are needed to indicate the necessary operations in a flowchart. These symbols are standardized by the American National Standard Institute (ANSI). These symbols are listed below:
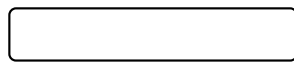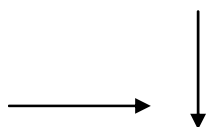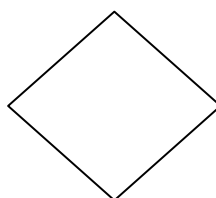
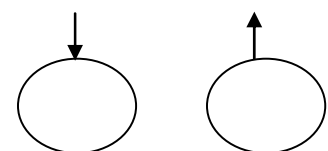Terminal
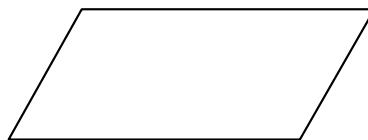
Input/Output
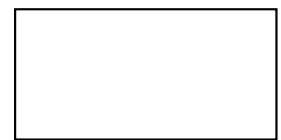
Processing

Flowlines

Decision

Connectors

<u>Terminal:</u> The terminal symbol, as the name implies is used to indicate the beginning (START), ending (STOP) and pauses (HALT) in the program logic flow. It is the first symbol and last symbol in the program logic. In addition, pause (HALT) used with a terminal symbol in program logic in order to represent some error condition.

<u>Input/Output:</u> The input/output symbol is used to denote any function of an input/output device in the program. If there is a program instruction to input data from a disk, tape, card-reader, terminal or any other type of input device, that step will be indicated in the flowchart with an input/output symbol. Similarly, all output instructions whether it is output on a printer, magnetic tape, magnetic disk, terminal screen or any output device, are indicated in the flowchart with an input/output symbol.

<u>Processing:</u> A processing symbol is used in a flowchart to represent arithmetic and data movement instructions. Thus all arithmetic processes of adding, subtracting, multiplying and dividing are shown by a processing symbol. The logical process of moving data from one location of the main memory to another is also denoted by this symbol. When more than one arithmetic and data movement instructions are to be executed consecutively, they are normally placed in the same processing box and they are assumed to be executed in the order of their appearance.

<u>Flowlines:</u> Flowlines with arrowheads are used to indicate the flow of operations, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right. Arrowheads are required only when the normal top to bottom flow is not to be followed. However, as a good practice and in order to avoid ambiguity, flowlines are usually drawn with an arrowhead at the point of entry to a symbol. Good practice also dictates that flowlines should not cross each other and that such intersections should be avoided whenever possible.

<u>Decision:</u> The decision symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternative points is possible. The criteria for making the decision should be indicated clearly within the decision box.

<u>Connector:</u> If a flowchart becomes very long, the flowlines start criss-cross at many places that causes confusion and reduces understandability of the flowchart. Whenever a flowchart becomes complex enough that the number and direction of flowlines is confusing or it spreads over more than one page, it is useful to utilize the connector symbol as a substitute for flowlines.

Problem:

A student appears in an examination that consists of total 10 subjects, each subject having maximum marks of 100. The roll number of the students, his name, and the marks obtained by him in various subjects is supplied as input data. Such collection of related data items that is treated as a unit is known as a record. Draw a flowchart for the algorithm to calculate the percentage marks obtained by the student in this examination and then to print it along with his roll number and name.
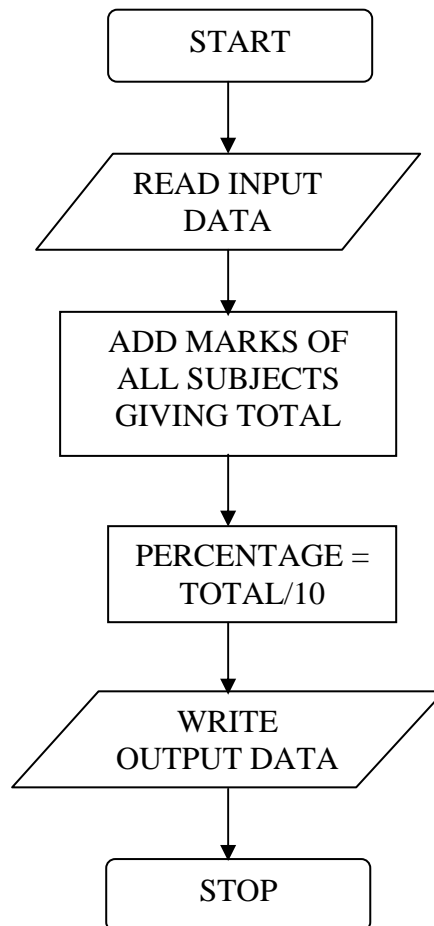
```
        ┌─────────────┐
        │    START    │
        └─────────────┘
               │
               ▼
       ╱─────────────╲
      ╱  READ INPUT   ╲
      ╲     DATA      ╱
       ╲─────────────╱
               │
               ▼
        ┌─────────────┐
        │ ADD MARKS OF│
        │ ALL SUBJECTS│
        │ GIVING TOTAL│
        └─────────────┘
               │
               ▼
        ┌─────────────┐
        │ PERCENTAGE =│
        │  TOTAL/10   │
        └─────────────┘
               │
               ▼
       ╱─────────────╲
      ╱    WRITE      ╲
      ╲ OUTPUT DATA   ╱
       ╲─────────────╱
               │
               ▼
        ┌─────────────┐
        │    STOP     │
        └─────────────┘
```

Fig : Flowchart

1.3 Coding:

In order to make a program in any programming language, what we have written is known as code. The act of writing code in a computer language is known as coding. In other words, code is a set of instruction that a computer can understand.

1.4 Compilation & Execution:

The process by which source codes of a computer (programming) language are translated into machine codes is known as compilation. After compilation if everything is ok, the code is going under other process that is known as execution. We can get the required output after execution process.

5

1.5 Debugging & Testing:

The process of finding and removing errors (also sometimes called buggs) from a program is known as debugging. One simple method of debugging is to place print statements throughout the program to display the values of variables. It displays the dynamics of a program and allows us to examine and compare the information at various points. Once the location of an error is identified and the error is corrected, the debugging statements may be removed.

Generally programmers commit three types of errors.They are
1. Syntax errors
2. Logic errors
3. Run-time errors

Syntax errors are those errors which are arised from violating the rules of programming language.On encountering these errors, a computer displays error message.It is easy to debug.Logic errors are those which arised when programmers proceed the logic process in wrong way or miss the some statements.It is difficult to debug such errors because the computer does not display them.Run-time errors are those which occur when programmers attempt to run ambiguous instructions.They occur due to infinte loop statement,device errors,sofware errors, etc.The computer will print the error message .Some of runtime errors are :

- Divide by zero
- Null pointer assignment
- Data over flow

Testing is the process of reviewing and executing a program with the intent of detecting errors. Testing can be done manually and computer based testing.

Manual Testing is an effecting error-detection process and is done before the computer based testing begins. Manual testing includes code inspection by the programmer, code inspection by a test group and a review by a peer group. Computer based testing is done by computer with the help of compiler (a program that changes source codes into machine codes word by word).

1.6 Program Documentation:

Program Documentation refers to the details that describe a program.While writng programs ,it is good programming practice to make a brief explanatory note on the programor program segment.This explanatory note is called comment.It explains how the program works and interact with it.Thus ,it helps other programmers to understand the program.There are two types of documentation:
1.Internal documentation
2.External documentation

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

6

Some details may be built-in as an integral part of the program. These are known as internal documentation. Two important aspects of internal documentation are; selection of meaningful variable names and the use of comments. Selection of meaningful names is crucial for understanding the program. For example,

<center>Area = Breadth * Length;</center>

is more meaningful than

<center>A = B * L</center>

And comments are used to describe actions parts and identification in a program. For example,

<center>/ * include file * / describes parts of program</center>
<center>/* header file * / describes parts of program.</center>

External documentation is an executable statement in a programIt may be message to the user to respond to the program requirement.This is accomplished using output statements .It makes a program more attractive and interactive.Some examples are:

print "Input the number one by one"
print "Do you want to continue ?"

Some Important Questions:

1. What is a Problem? What are basic steps in the process of Program Development? Explain each of them briefly.(PU2004)

2. Define algorithm and flowchart. What are the various symbols used to a flowchart? (PU 2003)

3. Differentiate between the flow chart and algorithm with the example. (PU2004)

4. What is debugging? (PU2005)

5. What is flow charting, describe its importance. (PU2006)

6. What is a flowchart? List the various commonly used flowchart symbols.

7. Why do we need documentation and testing for problem solving?(PU2006)

7

Chapter – 2

Introduction to C

C is a general-purpose, structured programming language. Its instructions consists of terms 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 lower level, thus bridging the gap between machine language and the more conventional high level language. This flexibility allows C to be used for system programming (e.g. for writing operating systems as well as for applications programming such as for writing a program to solve mathematical equation or for writing a program to bill customers). It also resembles other 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 Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell Laboratories in 1972. The new language was named C. Since it was 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 in C.

C was used mainly in academic environments for many years, but eventually with the release of C compiler for commercial use and the increasing popularity of UNIX, it began to gain widespread support among compiler professionals. Today, C is running under a number of operating systems including Ms-DOS. C was now standardized by American National 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 desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assemble language with the features of a high-level language 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 and powerful operators. It is many times faster than BASIC (Beginners All Purpose Symbolic Instruction Code – a high level programming language).

There are only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs. C is highly portable. This means that C programs written for one computer can be seen on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C Language is well suited for structure programming thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own function to the C library. With the availability of a large number of functions, the programming task becomes simple.

2.3 Basic Structure of C programs:

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

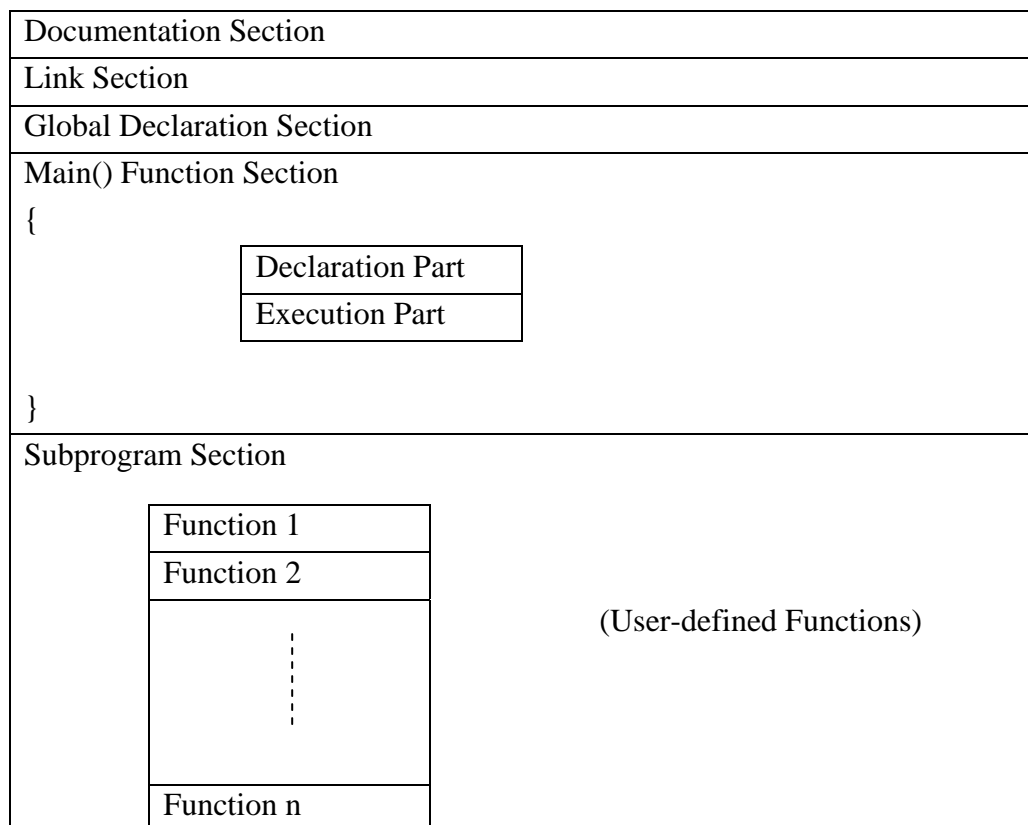| Documentation Section |
|---|
| Link Section |
| Global Declaration Section |
| Main() Function Section
{

         Declaration Part
         Execution Part

} |
| Subprogram Section

     Function 1
     Function 2   (User-defined Functions)
     |
     |
     Function n |

Fig : Basic Structure of a C program

The documentation section consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. The link section provides instructions to the compiler to link function from the system library. The definition defines all the symbolic constants. There are some variables that are used in more than one function. Such variables are called global variables and are declared in global declaration section that is outside of all the function.

Every C program must have one main( ) function section. This section consists two parts: declaration part and executable part. The declaration part declares all the variables used in the executable part. These two parts must appear between the opening and the closing braces. 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 the statements in the declaration and executable parts ends with a semicolon.

The subprogram section contains all the user-defined functions that are called in the main ( ) function. User-defined functions are generally placed immediately after the main ( ) function, although they may appear in any order. All section, except the main ( ) function section may be absent when they are not required.

Written by Er. Arun Kumar Yadav, Lecturer, Eastern College of Engineering Biratnagar

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-document mode. The file name should end with the characters ".c" like program .c, lab1.c, etc. Then the command under Ms DOS operating system would load the program stored in the file program .c i.e.

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 should then be corrected and compiled again.

The linking is done by the command

LINK program.obj

which generates the executable code with the filename program.exe. Now the command

.program

would execute the program and give the results.

```
/*        First Program written in C      */
  /*        Save it as hello.c                  */

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

        Print ("Hello, World \n");      /* statement */
                        }
 output : Hello, World

 /* Program to calculate the area of a circle */
 /* area.c */
 # include <stdio.h>     /* library file access */
   main( )                      /* function heading */
 {
   float radius, area;              /* variable decleration */
   clrscr();/* Console function :Used  to clear screen */
   printf ("Enter radious?=")   /* output statement */
   scanf ("%f", & radius);      /* input statement */
   area = 3.14159 * radius*radius ;    /* assignment statement */
   printf ("Area=%f", area);   /* output statement */
   getch();/* Console function :Used  to hold the same  screen */
 }
```

Some Important Questions:

1."C language is a middle level language". Explain this statement?(PU2008)
2.Write down the importance of C.(PU2004)
3.How can you define the structure of C -program? Explain with general form.(PU2003)

Chapter – 3

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 <u>data</u> consisting of numbers, characters and strings and to provide useful output known as <u>information</u>. The task of programming of data is accomplished by executing a sequence of precise instruction called a <u>program</u>. 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:

+ - * / = % & # ! ? ^ " ' ~ \ | < > ( ) [ ] { } : ; . , -
<u>(Blank space)</u>   <u>(Horizontal tab)</u>
     (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:

<u>C Tokens:</u>

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.: ( )   [ ]   { }

Computer Programming in C

<u>Identifiers:</u>

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   name   area   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)

<u>Keywords:</u>

There are certain reserved words, called keywords that have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used 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	struct
switch	typedef	union	unsigned	void
volatile	while			

    13

The keywords are all lowercase. Since upper and lowercase characters are not equivalent, 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 of which may be represented differently within the computer memory. There are three cases of data 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 known as built in data types. The basic data types are listed below. 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 has following classes: signed, unsigned, short, 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 has no values.

3.4   Constants, Variables

Constants in C refer to fixed values that do not change during the execution of a program. There are four basic types of constants in C. They are integer constants, floating point constants, character constants and string constants.

Integer and floating point constants represent numbers. They are often referred to collectively as numeric _ type constants. The following rules apply to all numeric type constants.

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 an operator that changes the sign of a positive constant though it can be thought of as a part 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.

<u>Integer Constants:</u>

An integer constant is an integer-valued number. Thus it consists of a sequence of digits. Integer (number) constants can be written in three different number systems: decimal (base 10), octal (base 8) and hexadecimal (base 16). Beginning programmers rarely however use anything other than decimal integer constants.

A decimal integer constant can consists of any combination of digits taken from the set 0 through 9. If the constant contains two or more digits, the first digit must be something other 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 (blank space)
123-45-6743	Illegal character (-)
0900	the first digit cannot be zero.

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

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 followed by any combination of digits taken from the sets 0 through 9 and a through f (either upper or lower case). The letters a through f (or A through F) represent the (decimal) quantities 10 through 15 respectively. Several valid hexadecimal integer constants are shown below:

0x	0X1	0X7FFF	0xabcd

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

0X12.34	Illegal character (.)
013E38	Doesn't begin with 0x or 0X.
0x.4bff	Illegal character (.)
0XDEFG	Illegal character(G)

Unsigned and Long Integer Constants:

Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of l, though they may not be negative. An unsigned integer constant can be identified by appending the letter ( ) (either upper or lowercase) to the end of the constant.

Long integer constants may exceed the magnitude of ordinary integer constants, but require more memory within the computer. A long integer constant can be identified by appending 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 the constant. The letters may be written in either upper or lowercase. However, the U must preced 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 an exponent (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 decimal point)
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 point constants:

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 quotation marks).

Several character constants are shown below:

‘A’   ‘X’   ‘3’   ‘?’   ‘ ´

Character constants have integer values that are determined by the computer's particular character 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 feature eliminates the dependence of a C program on a particular character set.

Most computers, and virtually all personal computer make use of ASCII (i.e. American Standard Code for Information Interchange) character set, in which each individual character is numerically encoded with its own unique 7-bit combination (hence a total of $2^7 = 128$ difference characters).

Several character constant and their corresponding values, as defined by ASCII character set are shown below:

Constant	Value
'A'	65
'X'	120
'3'	51
'?'	63
' '	32

These values will be the same for all computer 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"   "Washinton, 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 the newline characters that are embedded within the string. Thus, the string would be displayed as

Line 1

Line 2

Line 3

The compiler automatically places a null character (\0) at the end of every string constant, as the last character within the string (before the closing double quotation mark). This character is not visible 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 a single character string constant does not have an equivalent integer value and in fact, consists of 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 information within a designated portion of the program. In its simplest form, a variable is an identifier that is used to represent a single data item, i.e., a numerical quantity or a character constant. The data item must be assigned to the variable at some point in the program. A given variable can be assigned different data items at various places within the program. Thus, the information represented by the variable can change during the execution of the program. However, the data type associated with 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 integer variables, and that d is a character type. Thus, a, b and c will each represent an integer-valued quantity, and d will represent a single character. The type declaration will apply throughout the program.

The next four lines cause the following things to happen: the integer quantity 3 is assigned to a, 5 is assigned to b and the quantity represented by the sum a+b (.e. 8) is assigned to c. The character 'a' is assigned then assigned to d.

In the third line within this group, the values of the variables a and b are accessed simply 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 4 is assigned to a, replacing the earlier value, 3; then 2 is assigned to b, replacing the earlier value, 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 variables must be declared before they can appear in executable statements.

A declaration consists of a data type, followed by one or more variable names, ending with 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 the array.

19

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 floating variables, flag is a char-type variable and text is an 80-element, char-type array. Square brackets 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
```

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

Thus, c is an integer variable whose initial value is 12, star is a char type variable initially assigned the character '*', sum is a floating point variable whose initial value is 0. , and factor is double precision variable whose initial value is 0.21023×10^6.

\#   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 10 elements will represent the 10 characters within the word California, and the 11th element will represent the 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 the size is too large e.g.
   char text [20] = "California" ;
   the extra array elements may be assigned zeros, or they may be filled with meaningless characters.

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

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

Subscript: 0   1   2   3   4   5   6   7   8   9   10

3.6   Escape Sequence:

Certain nonprinting character, as well as the backslash (\) and apostrophe (`), can be expressed in terms of escape sequences. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, a linefeed (LF), which is

referred to as a newline in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters.

The commonly used escape sequences are listed below:

Character	Escape Sequence	ASCII Value
bell (alert)	\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 quotation mark respectively.

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

The general form '\000' represents an octal digit (0 through 7). The general form of a hexadecimal escape sequence is \xhh, where each h represents a hexadecimal digit (0 through 9 and a through f).

3.7 Preprocessors Directives:

The C preprocessor is a collection of special statements, called directives, that are executed at the beginning of compilation process. Preprocessors directives usually appear at the beginning of a program. A preprocessor directive may appear anywhere within a program. Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives #define and #include to a limited extent. 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 a predefined string composed of one or more tokens. The general format

#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 a program, we use conditional compilation. For that, we use following preprocessor directives such as #ifdef, #endif, #ifndef, #if .#else.

3.8   Typedef Statement:

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used 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 given to the data type. The existing data type may belong to any class of type, including the user defined

23

ones. The new type is new only in name, but not the data type. typedef cannot create a new type. Some examples of type definition are:

    typedef int units ;
    typedef float marks ;

    where, units represent int and marks represents float. They can be later used to declare variables as follows:

    units batch1, batch2 ;
    marks name1 [50] , name2[50] ;

batch1 and batch2 are declared as int variable and name1[50] and namer[50] are declared as 50 element floating point array variables. The main advantage of typedef is that we 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. The characters may represent numeric constant, a character constant or a string constant. Thus, a symbolic constant allows a name to appear in place of a numeric constant, a character constant or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

Symbolic constants are usually defined at the beginning of a program. The symbolic constants may then appear later in the program in place of the numeric constants, character constant, 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 text represents the sequence of characters that is associated with the symbolic name. It doesn't require 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 ;

Some Important Questions:

1. What do you mean by Character set in C?

2. What are constants and variables? Describe its types.(PU2006)

3. What is Symbolic constant, differentiate between keywords and identifier. (PU2006)

4. Describe different data types are used in c-program.

5. What is Escape Sequence? Write any four escape sequence with meaning and symbols.

6. What do you mean by a variable and a constant? List out any three-escape sequence with their uses. (PU2003)

7. Define variables and constant. What are the token of C language? (PU2005)

25

Chapter – 4

Operators & Expressions

Individual constants, variables, array elements and function references can be joined together by various operators to form expression. C includes a number of operators which fall into several different categories such as arithmetic operators, unary operators, rational and logical operators, assignment operators and the conditional operators, bitwise operator.

The data items that operators act upon are called operands. Some operators require two operands, while other act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variable as operand.

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) to carry out exponential.

The operands acted upon by arithmetic operators must represent numeric values. The remainder operator (%) requires that both operands be integers and the second operand be non 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. The operation always result in a truncated quotient (i.e. the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, 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 are grouped hierarchically according to their precedence. Operations with a high precedence are

carried out before operation having a lower precedence. The natural order of evaluation can be 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 are carried 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 subtraction operations are carried out from left to right, as are consecutive multiplication, division and remainder 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, assuming that 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, assuming that 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, and depending 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 the arithmetic 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	1

In addition to the relational and equality operators, C also includes the unary operator ! that negates the value of a logical expression, i.e. it causes an expression that is originally true to become false and vice-versa. This operator is referred to as the logical negative (or logical not) operator.

Logical operators:

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

\# 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'. Several complex 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') || (i+f<=10)	true	1
!(i>(f+1))	false	0

Assignment Operators:

There are several different assignment operators in C. All of them are used to form assignment expressions which assign the value of an expression to an identifier.

The most commonly used assignment operator is =. Assignment expressions that make use of the operator are written in the form

identifier = expression

where identifier generally represents a variable, and expression represents a constant, a variable 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. The assignment operator   is used to assign a value to an identifier, where as the equality operator is used to determine if two expressions have the same 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 known as 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 to write.
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 known as unary operators. Unary operators usually precede their single operands, though some unary operators are written after their operands.The common use unary operators in C are:

1. Increment Operator (+ +)
2. Decrement Operator (- -)
3.Unary minus(-)
4.unary plus(+)

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

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

The increment 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 before the operand then it is called as prefix unary operator. If the operator is written after the operand, then it is called as postfix unary operator. When prefix is used, the operand will be altered 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.

The most common use unary operator is unary minus, where a numeric constant, variable expression is preced by a minus sign.Unary minus is distinctly difference from the arithmetic operator which denotes subtraction (-).The subtractor operator requires two separate operands .Several examples of unary minus are

-743   -0x7fff   -0.2   -5E-8   -root1   -(x+y)   -3*(x+y)

In C, all numeric constants and variable are considered as positive. So, unary plus is not written as unary minus such as

+743 = 743

+(x+y) = x+y

\#   A C program includes an integer variable i whose initial value is 1. Suppose the program 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 or unary minus or plus is same and associativity is right to left.

Bitwise Operators:

Bitwise Operators are used for manipulating data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators can be applied only to integer type signed or unsigned) and not to float or 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, pointer operator (* and &) and member selection operators. (.and ➤)

Comma Operator:

The comma operator can be used to link the related expression together. A comma linked list of expression are evaluated left to right and the value of right-most expression is the 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. It is 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 e.g.

\# Suppose that i is an integer variable, x is a floating-point variable, d is double-precision variable 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) ;
```

might generate the following output :

```
integer  :2
float  :4
double :8
character  :1
```

The above statement can also be written as

```
printf("integer :%d\n",sizeof(int));
and so on.
```

\# char text[ ]="Kathmandu"

```
printf("Number of characters =%d",sizeof(text));
```

will generate the following output:

Number of characters =10

Conditional Operators:

The operator ?: is known as conditional operator. Simple conditional operations can be carried out with conditional operator. An expression that make use of the conditional operator is called a conditional expression. Such an expression can be written in place of traditional if-else statement.

A conditional expression is written in the form:

expression1 ? expression2 : expression3

When evaluating a conditional expression, expression1 is evaluated first. If expression1 is 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 the if_else statement as follows:

if (a<b)

  x = a ;

else

  x = b ;

Summary of C operators:

Operators	Description	Associativity	Precence Rank
( )	Function call	Left -> Right	1
[ ]	Array element reference		
+	Unary plus	Right -> Left	2
-	Unary minus		
++	Increment		
--	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference		
&	Address		
sizeof	Sizeof operator		
Operators	Description	Associativity	Precence Rank
*	Multiplication	Left ->Right	3
/	Division		
%	Modulus		
+	Addition	Left ->Right	4
-	Subtraction		
<<	Leftshift	Left ->Right	5

35

>>	Rightshift		
 <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left ->Right	6
== !=	Equality Inequality	Left ->Right	7
&	Bitwise AND	Left ->Right	8
^	Bitwise XOR	Left ->Right	9
|	Bitwise XOR	Left ->Right	10
&&	Logic AND	Left ->Right	11
||	Logic OR	Left -> Right	12
?:	Conditional operator	Right -> Left	13
= += -= *= /= %=	Assignment operators	Right -> Left	14
,	Comma operator	Left -> Right	15

Some Important Questions:

1. Write the general form of ternary operator and explain with example. (PU2003)

2. What is a datatype? Write down the various types of it in C-program. (PU2003Back)

3. What are the various types of operators used in C language? Give a table that shows their precedence and associatively. (PU2005)

4. Define the data types. Describe different data types. (PU2005Back)

5. Write a program to enter two numbers. Make the" comparison between them with conditional operator. If the first number is greater than second, perform multiplication otherwise division.(PU 2007)

6. What are operators? Describe unary, binary and ternary operators with examples.

<div style="border:1px solid black">

Chapter – 5
</div>

Input and Output Operations

A program is a set of instructions that takes some data and provides some data after execution. The data that is given to a program is known as input data. Similarly, the data that is provided by a program is known as output data. Generally, input data is given to a program from a keyboard (a standard input device) or a file. The program then proceeds the input data and the result is displayed on the screen (monitor – a standard output device) or a file. Reading input data from keyboard and displaying the output data on screen, such input output system is considered as conio input out.
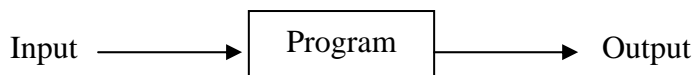
Input ⟶ | Program | ⟶ Output

Fig: Input/ Output

To perform input/output operation in console mode, C has a number of input and output functions. When a program needs data, it takes the data through the input functions and sends the results to output devices through the output functions. Thus the input/output functions are the link between the user and the terminal.

As keyboard is a standard input device, the input functions used to read data from keyboard are called standard input functions. The standard input functions are scanf( ), getchas( ), getch( ), gets( ), etc. Similarly, the output functions which are used to display the result on the screen are called standard output functions. The standard output functions are printf( ), putchas( ), putch( ), puts( ), etc. The standard library stdio.h provides functions for input and output. The instruction #include<stdio.h> tells the compiler to search for a file named stdio.h and place its contents at this point in the program. The contents of the header file become part of the source code when it is compiled.

5.1   Types of I/O

The input/output functions are classified into two types:
i. Formatted functions
ii. Unformatted functions

Formatted Functions:
Formatted functions allow the input read from the keyboard or the output displayed on screen to be formatted according to our requirements. The input function scanf() and output

37

function: printf() fall under this category. While displaying a certain data on screen, we can specify the number of digits after decimal point, number of spaces before the data, the position where the output is to be displayed, etc, using formatted functions.

Formatted Input:

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data.

         20   11.23   Ram

The above line contains three types of data and must be read according to its format. The first be read into a variable int, the second into float, and the third into char. This is possible in C using the scanf function. scanf( )stands for scan formatted.

The input data can be entered into the computer from a standard input device keyboard by means of the C library function scanf. This function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully. The general syntax of scanf function is

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

where,

control string refers to a string containing certain required formatting information so also known as format string and arg1, arg2, ……, argn are arguments that represent the individual input data items. Actually, the arguments represent pointers that indicate the addresses of the data items within the computer's memory.

The control string consists of individual groups of characters, with one character group for each input data item. Each character group must begin with a percent sign (%). In its simplest form, a single character group will consist of the percentage sign, followed by a conversion character which indicates the types of corresponding data item. Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace (i.e. blankspace), tabs or newline characters. If whitespace characters are used to separate multiple character groups in the control string, then all consecutive white-space characters in the input data will be read but ignored. The use of blank spaces as character group separators is very common.

Several of the more frequently used conversion characters are listed below:

Conversion Characters	Meanings
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating-point value
f	Data item is a floating-point value
g	Data item is a floating point value

h	Data item is short integer
i	Data item is decimal, hexadecimal or octal integer
o	Data item is octal integer
s	Data item is a string followed by a white space character (the null character (\O) automatically be added at the end
u	Data item is an unsigned decimal integer
x	Data item is a hexadecimal integer
[…]	Data item is string which may include white space
l	Data item is long integer or double
L	Data item is long double

Table: Conversion Charater Table

Example: 1

```
#inlcude<stdio.h>
    main( )
    {        char item[20] ;
             int partno ;
             float cost ;
                   - - - - - -
             scanf ("%s %d %f", item, &partno, &cost) ;
                   - - - - - - -
                              }
```

The following data items could be entered from the standard input device when the program is executed.

Different methods for entering the inputs:

Biratnagar   1245   0.05   ↵

Biratnagar   ↵
    1245   ↵

    0.05   ↵
Biratnagar ↵
    1245   0.05 ↵

Biratnagar   1245 ↵

39

0.05 ↵

Example: 2

```
#include<stdio.h>
main( )
{
    char line [80];
    - - - - - -
    scanf ("%[  ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line) ;
    - - - - - -
}
```

If the string

EASTERN COLLEGE OF ENGINEERING

Is entered from the standard input device when the program is executed, the entire string will be assigned to the array line since the string is comprised entirely of uppercase letters & blank spaces.

If the string were written as

Eastern College of Engineering

then only the single letter E would be assigned to line. Since the first lowercase letter (in this case a) would be interpreted as the first character beyond the string.

Example: 3

```
#include<stdio.h>
    main( )
    { char    line [80];
    - - - - -
    scanf ("   %[^\n]", line) ;
    - - - - -
    }
```

A variation of this feature which is often more useful is to precede the characters within the square brackets by a circumflex (or caret). If the character within the brackets is simply the circumflex followed by a newline character, then string entered from the standard input device can contain any ASCII characters except the newline characters (line feed). Thus, the user may enter whatever he or she wishes and then presses the Enter Key. The Enter Key will issue the newline character, thus signifying the end of the string.

The consecutive nonwhitespace characters that define a field. It is possible to limit the number of such characters by specifying a maximum field width for that data item. To do so, an unsigned integer indicating the field width is placed within the control spring between the percent sign (%) and the conversion character.

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

40

The data item may contain fewer characters than the specified filed width. However, the number of characters in the actual data item cannot exceed the specified field width. Any characters that extend beyond the specified filed width will not be read. Such leftover characters may be incorrectly interpreted as the components of the next data item.

Example: 4

```
#include<stdio.h>
  main( )
  {
   int a, b, c ;
   - - - - -
   scanf ("%3d %3d %3d", &a, &b, &c) ;
   - - - - -
   }
```

Suppose the input data items that are entered as

  1 2 3 ↵

Then the following assignment will result:

  a = 1, b = 2, c = 3

If the data had been entered as

  123   456   789

Then the assignment would be

  a = 123, b = 456, c = 789

Now suppose that the data had been entered as

  123456789

Then the assignments would be

  a = 123, b = 456, c= 789

Finally, suppose that the data had been entered as

  1234   5678   9

The resulting assignments would now be

  a = 123   b = 4   c= 567

Example: 5

```
#include<stdio.h>
  main( )
  { int  i ;
    float x ;
    char c ;
    - - - - -
    scanf ("%3d %5f  %c", &i, &x &c) ;
```

41

```
       - - - - -
         }
```
If the data items are entered as

        10   256.875   T

The output would now be

        10, 256.8, 7

The remaining two input characters (5 and T) will be ignored.

Example:6


```
#include<stdio.h>
     main( )
     {
        short ix, iy ;
        long lx, ly ;
        double dx, dy;
        - - - - - -
        scanf ("%hd %ld %lf", &ix, &ly, &dx) ;
        - - - - - -
     }
```

The control string in the first scanf function indicates that the first data item will be assigned to a short decimal integer variable. The second will be assigned to a long decimal integer variable, and the third will be assigned to a double precision variable.

The control string in the second scanf function indicates that the first data item will have a maximum field width of 3 characters and it will be assigned to short octal integer variable, the second data item will have a maximum field width of 7 characters and it will be assigned to a long hexadecimal integer variable, and the third data item will have a maximum field width of 15 characters and it will be assigned to double precision variable.

In most version of C, it is possible to skip over a data item, without assigning it to the designated variable or array. To do so, the % sign within the appropriate control group is followed by an asterisk (*). This feature is referred to as assignment suppression.

Example: 7
```
#include<stdio.h>
     main( )
{
     char item[20] ;
```

        int partno ;
        float cost
        - - - - - -
        scanf ("%s %*d %f", item, &partno, &cost),
        - - - - - -
}

    If the corresponding data item are input
        fasterner   12345   0.05

    fasterner is assigned to item and 0.05 will be assigned to cost. However 12345 will not be assigned partno because of asterisk, which is interpreted as an assignment suppression character.

Formatted Output:

    Formatted output refers to the output of data that has been arranged in a particular format. The printf() is a built-in function which is used to output data from the computer onto a standard output device i.e. screen, This function can be used to output any combination of numerical values, single character and strings. The printf() statement provides certain features that can be used to control the alignment and spacing of print-outs on the terminals. The general form of printf() statement is printf (control string, arg1, arg2, …. , argn) where control string refers to a string that contains formatting information, and arg1, arg2, ……., argn are arguments that represent the individual output data item. The arguments can be written as constants, single variable or array names, or more complex expressions. Function references may also be included. In contrast to scanf() function, the arguments in a printf() function do not represent memory addresses and therefore are not preceded by ampersands. An individual character group in control string will consist of the percent sign, followed by a conversion character indicating the type of the corresponding data item.

    Multiple character group can be contiguous, or they can be separated by other characters, including whitespace character. These "other" characters are simply transferred directly to the output device, where they are displayed. The use of blank spaces are character group separators is particularly common.

Several of the more frequently used conversion characters are listed below:

Conversion Character	Meaning
c	Data item is displayed as a single data
d	Data it Data item is displayed as floating point
e	value with an exponent em is displayed as a signed decimal
f	Data item is displayed as floating point without an exponent
g	Data item is displayed as floating point value using either e-type or f-type conversion
i	Data item is displayed as a signed decimal integer
o	Data item is displayed as an octal integer w
s	Data item is displayed as a string ithout a leading zero
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer without the leading 0x.

Table: Conversion Charater Table

Note :l for long int, h for signed/unsigned short, L for double.

For example: 1
```
#include<stdio.h>
    main( )
    {
      char item [20] ;
      int partno ;
      float cost ;
      - - - - - -
      printf ("%s%d%f ", item, partno, cost) ;
    }
```
Suppose fastener, 12345 and 0.5 have been assigned to name, partno and cost. So, the output generated will be

fastener123450.05

Example: 2
```
#include<stdio.h>
    main( )
    {
```

```
double x = 5000.0, y = 0.0025;
printf ("%f%f%f%f%f\n\n", x, y, x*y, x/y) ;
printf ("%e%e%e%e, x, y, x*y, x/y) ;
}
```
The output are

5000.000000   0.002500   12.500000   2000000   .000000

5.000000e+03   2.5000000e-03   1.250000e+01   2.000000+06

Example: 3
/* read and write a line of text */
#include <stdio.h>

```
main( )
{
  char line [80] ;
  scanf ("%[^\n]", line) ;
  printf ("%s",  line) ;
}
```
Arun Kumar ↵
Arun Kumar

Control string:

The general syntax of control string :

%[Flag] [Field width] [.precision] conversion character

➤ Flags [Optional]

The flag affect the appearance of the output. They must be placed immediately after the percent sign. The flags may be -, +, 0, blank space or #.

Flags	Meanings
-	: Data item is left justified within the field. The blank spaces required to fill the

	minimum field width will be added after the data item rather than the data item.
+	: A sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are precededby a sign.
0	: Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item
(blank space)	: A blank space will precede each positive signed numeric data item .This flag is overwritten by the + flag if both are present.
#	: When # flag is used with %0 or %x, it causes octal and hex items to be preceded by 0 and 0x respectively. When it is used with %e, %f or %g, it causes a decimal point to be presented in all floating point numbers, even if it is a whole number.

Table:Flags

> Field Width [Optional]

The field width is an integer specifying the minimum output 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.

> Precision [Optional]

The operation of precision field depends on the types of conversion. It must start with a period (.)

Format for Integer Output:
%wd

where w is the integer number specifying the minimum field width of output data. If the length of the variable is less than the specified field width, then the variable is right justified with leading blanks.

For example : integer number

     i.e. int n = 1234

Format	**Output**					

printf ("%d", n) ;

1	2	3	4

l =w

printf ("%6d", n)

		1	2	3	4

w>l

printf ("%2d", n)

1	2	3	4

w<l

printf ("%-6d", n)

1	2	3	4		

w>l

printf ("%-6d", n)

0	0	1	2	3	4

Format for floating point output: The general form:

%w.pf

%w.pe

where w is the integer width including decimal point

p is the precision

f and e are conversion characters

Example :1

Format	**Output**			

printf ("%.4d", 12) ;

0	0	1	2

printf ("%.4O", 12) ;

0	0	1	4

printf ("%4x", 12)

0	0	0	c

printf ("%.2f", 12.3456) ;

1	2	.	3	4

printf ("%.2e", 12.3456) ;

1	.	2	3	e	+	0	1

printf ("%.4g", 12.3456) ;

1	2	.	3	5

#eg .:2
float x = 12.3456

Format   **Output**

printf ("%7.4f", x) ;

1	2	.	3	4	5	6

printf ("%7.2f", x) ;

		1	2	.	3	5

printf ("%-7.2f", x) ;

1	2	.	3	5		

printf ("%7.2f", -x) ;

	–	1	2	.	3	5

printf ("%6.4f", x) ;

1	2	.	3	4	5	6

printf ("%10.2e", x) ;

		1	.	2	3	e	+	0	1

printf ("%-10.2e", x) ;

1	.	2	3	e	+	0	1		

printf ("%10.2e", -x) ;

	–	1	.	2	3	e	+	0	1

Output of Strings:

The general form of control string is %w.p.s

where

w specifies the field width for display and p instructs that only the first P characters of the string are to be displayed. The display is right justified

For example

   char str[10] = "MY NEPAL"

Format	**Output**

printf ("%s", str) ;

M	Y		N	E	P	A	L

printf ("%10s", str) ;

		M	Y		N	E	P	A	L

printf ("%4s", str) ;

M	Y		N

printf ("%10.6s", str) ;

M	Y		N	E	P			

printf ("%4s", str) ;

M	Y		N	E	P	A	L

Unformatted Functions:

   Unformatted functions do not allow the user to read or display data in desired format. These library functions basically deal with a single character or a string of characters. The functons getchar( ), putchse( ), gets( ), puts( ), getch( ), getche( ), putch( ) are considered as unformatted functions.

getchar( ) and putchar ():

   The getchar( ) function reads a character from a standard input device. The general syntax is

      character_variable = getchar( );

  where character_variable is a valid C char type variable. When this statement is encounted, the computer waits until a key is pressed and assign this character to character_variable.

   The putchar( ) function displays a character to the standard output device. The general syntax of putchar( ) function is

      putchar(character_variable) '

  where character_variable is a char type variable containing a character.

```
/*Program to demonstrate unformatted function*/
/*unformat.c*/
#include<stdio.h>
#include<conio.h>
  main( )
{
  clrscr( )
  char gender ;
  printf ("Enter gender M or F :" ) ;
  putchar (gender) ;
getch( );
}
```

Output:

Enter gender M or F : M↵

Our gender is : M

Note:

clrscr( ) is a console fumction used to clear console(display) screen .clrscr( ) is pronoused clearscreen.getch( ) function is used to hold the console screen.

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

The functions getch( ) and getche( ) reads a single character the instant it is typed without waiting for the enter key to be hit. The difference between them is that getch( ) reads the character typed without echoing it on the screen, while getche( ) reads the character and echoes (displays) it on the screen. The general syntax of getch( ):

character_variable = getch( ) ;

Similarly, the syntax of getch( ) is

character_variable = getche( ) ;

The putch( ) function prints a character onto the screen. The general syntax is

putch(character_variable) ;

These three functions are defined under the standard library function conio.h and hence we should include this in our program using the instruction #include<conio.h>

#include<stdio.h>

```
#include<conio.h>
     main( )
     {
      char  ch1,  ch2 ;
      clrscr( );
      printf ("Enter first character:") ;
      ch1 = getch( ) ;
      printf ("\n Enter second character:") ;
      ch2 = getche( )
      printf ("\n First character:") ;
      putch(ch1) ;
      printf("\n Second character:") ;
      putch(ch2) ;


}
```
Output:

    Enter first character :

    Enter second character : b ↵

    First character : a

    Second character : b

> a ↵

Since the first input is taken using getch() function, the character 'a' entered is not echoed. However, using getche() function, we can see what we have typed. In both cases, input accepted as soon as the character typed. The last getch() simply takes a character but doesnot store it anywhere. So, its work is merely to hold the output screen until a key is pressed.

gets( ) and puts( ):

The gets( ) function is used to read a string of text containing whitespaces, until a newline character is encountered. It offers an alternative function of scanf( ) function for reading strings. Unlike scanf( ) function, it doesnot skip whitespaces. The general syntax of gets( ) is gets(string_variable) ;

The puts( ) function is used to display the string onto the terminal. The general syntax of puts( ) is

        puts (string _variable)

This prints the string value of string_variable and then moves the cursor to the beginning of the next line on the screen.

```
#include<stdio.h>
#include<conio.h>
    void main( )
    {
       char name[20] ;
       clrscr( );
       printf ("Enter your name:") ;
       gets (name) ;
       printf ("your name is:") ;
       puts (name) ;
       getch ();
    }
```
output:

  Enter your name : Ram Kumar ↵

  Your name is : Ram Kumar

Some Important Questions:

  1.Write a general form of input/output statement of C-programming.(PU2003)

  2.Find the output the following program: (PU2003)

```
#include<stdio.h>
Main( )
{
int a,b,c;
a = 155;
b=4239;
c=21234;
printf("%5d,%5d,%5d\n",a,b,c);
printf("%3d,%4d,%5d\n",a,b,c);
}
```

  3 .Find the output of the following program: (PU2004)

```
include<stdio.h>
#include<cnio.h>
void  main( )
{
int x = 10;
float y = 20;
char z = 'C';
clrscr( );
printf("%8d,%8.1f, %8e \n",x,y,z);
```

```
printf("%6d, %6.1f, %6c \n",x,y,z);
getch( );
}
```

4.Write short noteson :Formatted input/output function. (PU2004)

5.Distinguish between getc( ) a getchar().(PU2005)

6.What is difference between formatteI/O and unformatted I/O? (PU2005Back)

[b] Find the output of the following program .(PU2005Back)

```
#include<stdio.h>
#include<conio.h>
void main( )
{
char class [] ="country";
printf("%s",class);
printf("%c",class[61]);
printf("%s \r%c", class, class [6]);
getch( );
}
```

7.Write the various input unformatted function and describe any two.

8. Write the various output unformatted function and describe any two.

9. Write the input/output formatted function along with syntaxes..

Chapter – 6

Control Statements

The statements which alter the flow of execution of the program are known as control statements. In the absence of control statements, the instruction or statements are executed in the same order in which they appear in the program. Sometimes, we may want to execute some statements several times. Sometime we want to use a condition for executing only a part of program. So, control statements enable use to specify the order in which various instruction 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 statement several times. So, a loop may be defined as a block of statements which are repeatedly executed for a certain number of times or until a particular condition is satisfied. There are three types of loop statements in C:

1. For
2. While
3. Do...while

Each loop consists of two segments, one is known as the control statement and the other is the body of the loop. The control statement in loop decides whether the body is to be executed or not. Depending on the position of control statement in the loop, loops may be classified either entry_controlled loop or exit_controlled loop. While and For are entry_controlled loops where 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 of repetitions is known in advance, the use of this loop will be more efficient. Thus, this loop is also known as determinate or definite loop. The general syntax

```
for (counter initialization ; test condition ; increment or decrement)
{
  | * body of loop * |
}
```
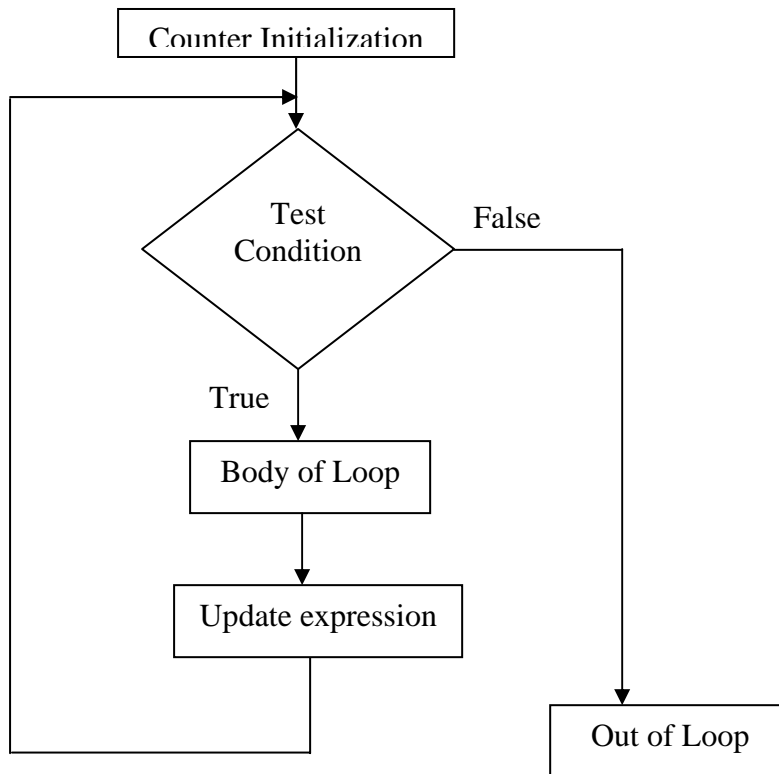
Flowchart of For Loop:

Fig : Flowchart of For Loop

Algorithm:

- Program to calculate factorial

1.Start.
2.Print" Enter a number whose factorial is to be calculated".
3.Read num.
4.Initilize fact to 1 and counter i to1
5.For i<=num
     fact=fact*i
             i++
End of For
6.Print fact as factorial of the number num.
7.Stop

55

For example:

```
/* Calculate the factorial of a number */
/* factorial.c */
#include<stdio.h>
        main( )
        {
                int num, i ;
                long fact = 1;
                clrscr( );
                printf ("\n 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 ) ;
                getch( );
        }
```

Output:

        Enter a number whose factorial is to be calculated : 5 ↵
        The factorial is : 120

While Loop:

        The while statement can be written as

```
            while(condition)                while(condition)
            statement ;                     {
                                              statement ; /* body of the loop */
                                              statement ;
                                              - - - - - - -
                                            }
```

First the condition is evaluated ; if it is true then the statements in the body of loop are executed. After the execution, again the condition is checked and if it is found to be true then again the statements in the body of loop are executed. This means that these statements are executed continuously till the condition is true and when it becomes false, the loop terminates and the control comes out of the loop. Each execution of the loop body is known as iteration.
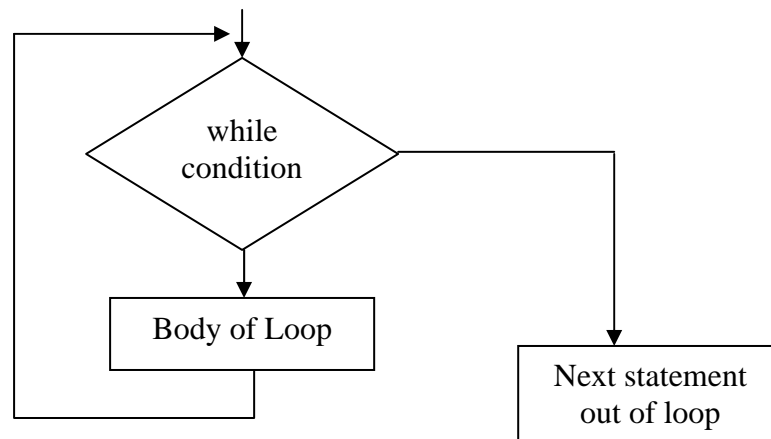
fig : Flowchart of while loop

/* Program to print the sum of digits of any num */

```c
#include<stdio.h>
      main( )
      {
        int n, sum = 0, rem ;
              clrscr( );
        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 ;        /*n = n/10 */
                               /* skipping last digit */

      }/*end of while*/
   printf ("Sum of digits = %d \n", sum) ;
         getch( );
        }
```

Output:
        Enter the number : 1452
        Sum of digits = 12

do...while loop:

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

57

```
do                              do
   statement ;                  {
   while(condition) ;                     statement1 ;
                                          statement2 ;
                                          - - - - - - - -
                                          statementn ;
                              } while(condition) ;
```
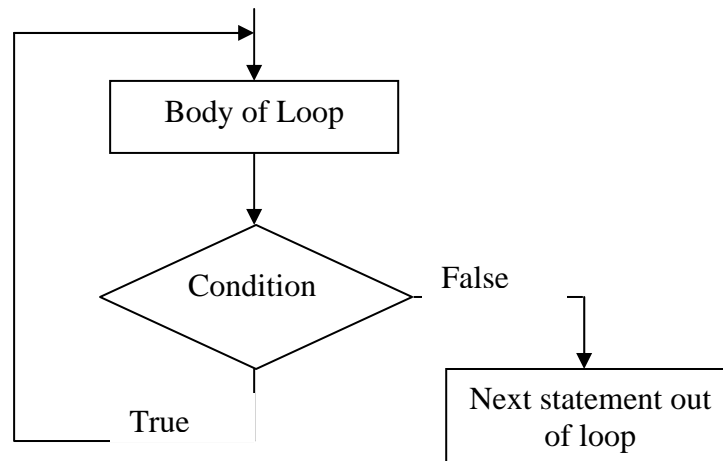
Flowchart of do...while is:

fig : flowchart of do...while loop

Here firstly the segments inside the loop body are executed and then the condition is evaluated. If the condition is true, then again the loop body is executed and this process continues until the condition becomes false. Unlike while loop, here a semicolon is placed after the condition. In a 'while' loop, first the condition is evaluated and then the statements are executed whereas in do while loop, first the statements are executed and then the condition is evaluated. So, if initially the condition is false the while loop will not execute at 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 ;clrscr( );
          do
          {printf ("%d\t", i) ;
                  i++ ;
          } while (i<=10) ;
           printf("\n") ; getch( );
        }
```
Output :   1 2 3 4 5 6 7 8 9 10

Differences between while loop and do while loop:

while	do...while
1. while loop is entry controlled loop i.e. test condition is evaluated first and body of loop is executed only if this test is true.	1. do...while loop is exit controlled loop i.e. the body of the loop is executed first without checking condition and at the end of body of loop, the condition is evaluated.
2. The body of the loop may not be executed at all if the condition is not satisfied at the very first attempt.	2. The body of the loop is always executed at least once.

3. syntax :
 while (condition)
  {
    body of the loop
  }
	3. syntax :
	  do
	  {
	     body of the loop
	  } while (condition)

4. Draw flowchart	4. Draw flowchart

Nesting of loops:

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

```
/* program to understand nesting in for loop * /
#include<stdio.h>
    main( )
    {
      int i,j ;
    clrscr( );
      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") ;
                    }  getch( ); }
```

59

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 the category of control statements. Following are decision making statements:

- if statements
- if....else statements
- else if statement
- Nested if...else statement
- switch statement

if statement:

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. This is a bi-directional condition control statements. This statement is used to test a condition and take one of two possible actions, If the condition is true 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 the program). 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 ;
                                         }
```

There can be a single statement or a block of statements after the if part.

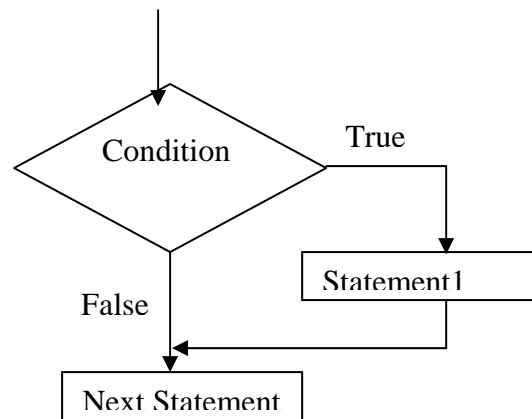Flowchart:

Computer Programming in C

Fig : Flowchart of if control statement

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

For eg:

|* ### Program to check whether the number is –ve or +ve ### *|

```
#include<stdio.h>
    main( )
    {
      int num ;
      clrscr( );
      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 there are two possible actions – one when a condition is true and the other when it is false. The syntax is :

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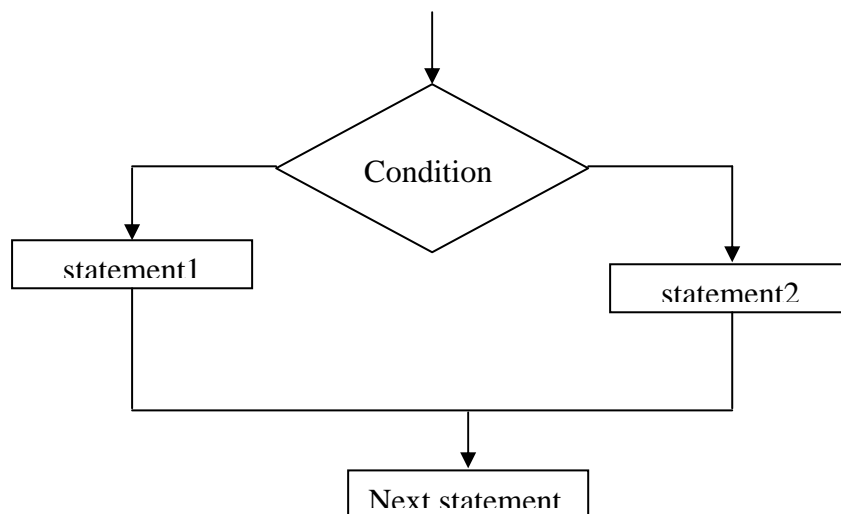
Flowchart:

Fig : Flowchart of if...else control statement

Here if the condition is true then statement1 is executed and if it is false then statement2 is executed. After this the control transfers to the next statement which is immediately after the if...else control statement.

```c
/* Program to check whether the number is even or odd */
#include<stdio.h>
    main( )
    {
      int num, remainder ;
      clrscr( );
      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") ;
        getch( );
         }
```

Output:

Enter a number : 15 ↵

Number is odd.

else if statement:

Nested if ...else statement:

We can have another if... else statement in the if block or the else block. This is called nested 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,
    clrscr( );
      printf ("Enter year: ") ;
      scanf ("%d", &year) ;
```

```
    if(year%100 = = 0)
     {
             if(year%400 = = 0)
               printf ("Leap year \n") ;
             else
               printf ("Not leap year\n") ;
     }
       getch( );
}
```

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 ;
            clrscr( );
            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) ;
            getch( );
    }       /* 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 except the last else part. This type of nesting is frequently used in programs and is also known as else if ladder.

```
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 ;
```
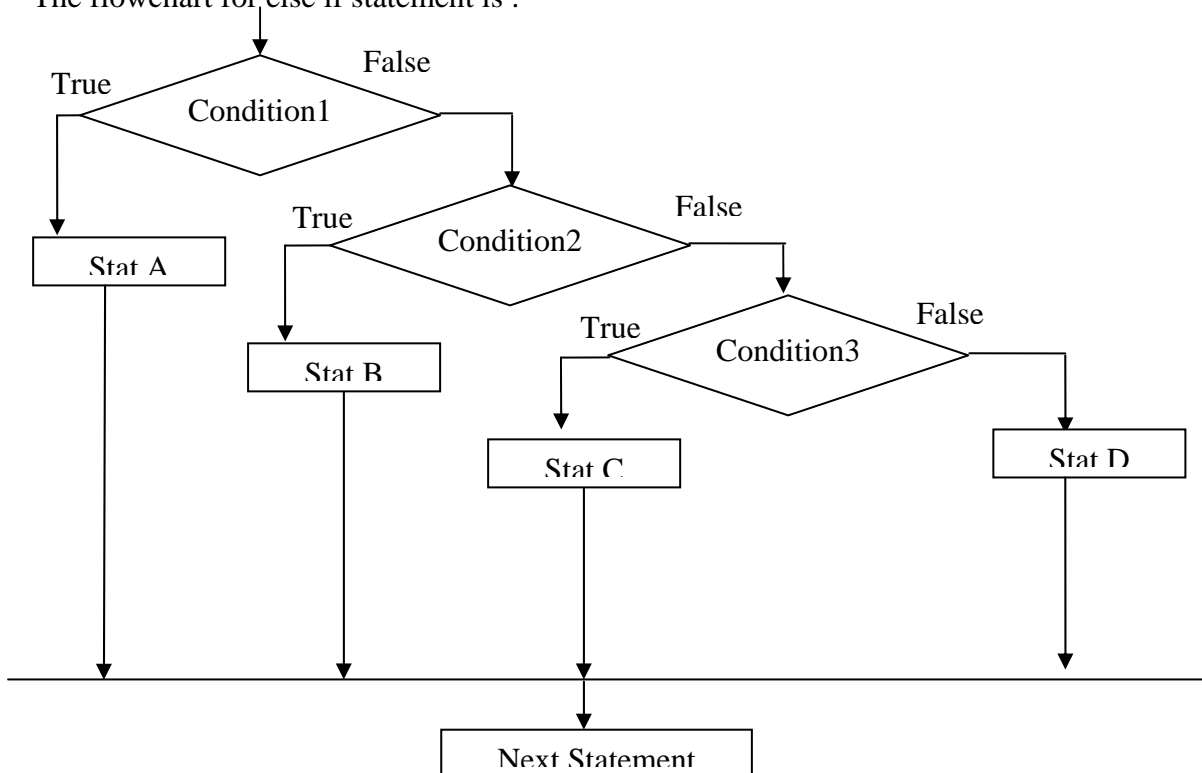
The flowchart for else if statement is :

Fig : Flowchart of else if ladder

65

/* 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 ;
        clrscr( );
        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) ;
        getch( );
    }
```

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 and moreover the conditions here are more lengthy.

6.3   Statements: switch, break, continue, goto

<u>Switch Statement:</u>

This is a multi-directional conditional control statement. Sometimes, there is a need in program to make choice among number of alternatives. For making this choice, we use the switch 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 switch keyword can be any C expression that yields an integer value or a character value. It can be value of any integer or character variable, or a function call returning on integer, or an arithmetic, logical, relational, bitwise expression yielding integer value.

The constants following the case keywords should be of integer or character type. These constants must be different from each other.

The statements under case can be any valid C statements like if...else, while, for or even another switch statement. Writing a switch statement inside another is called nesting of switches.

Firstly, the switch expression is evaluated then value of this expression is compared one by 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 constant matches with the value of the expression then the block of statements under default is executed.
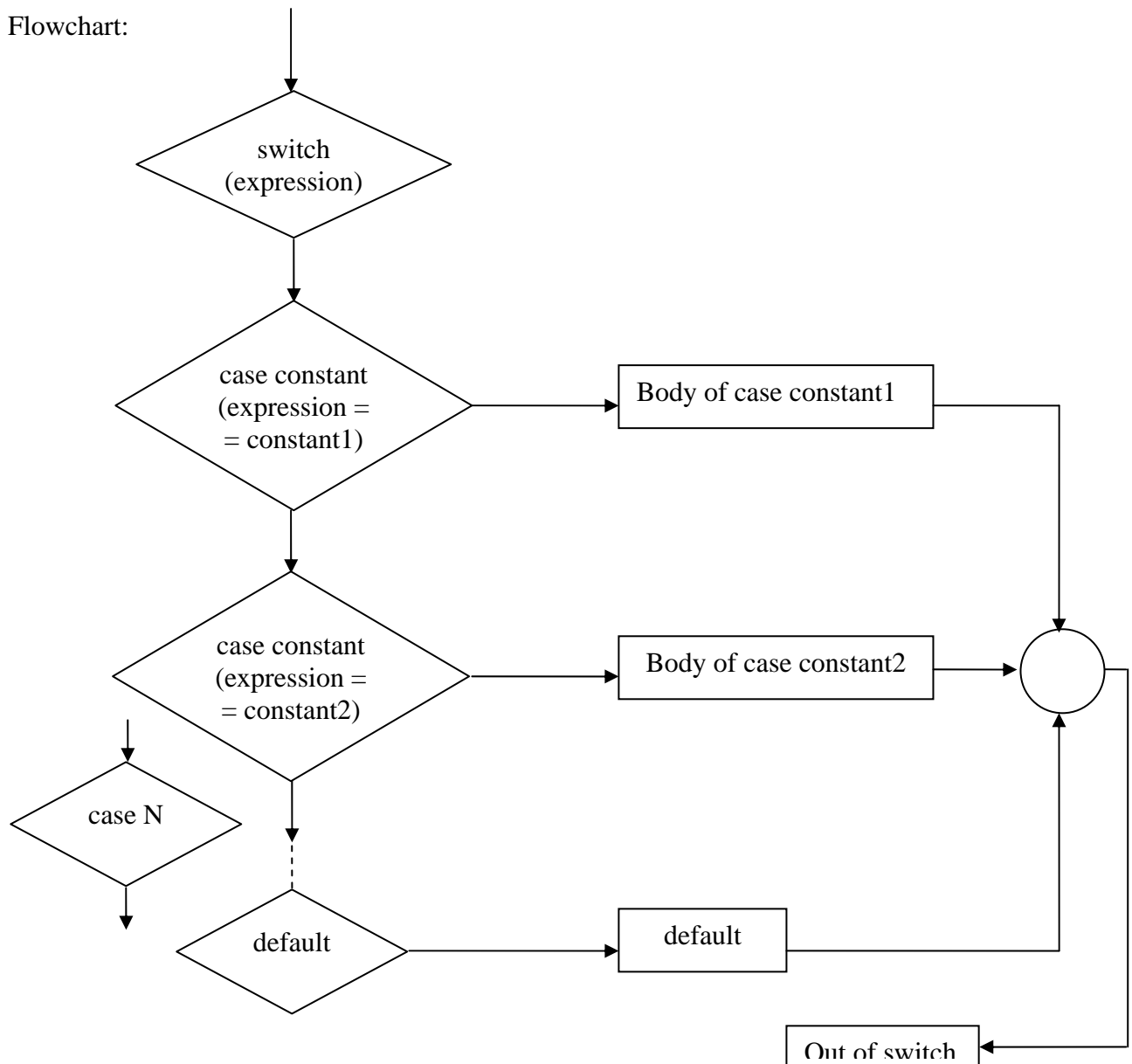
67

Flowchart:

fig : Flowchart of switch statement

```
/* Program to understand the switch control statment */
#include<stdio.h>
    main( )
    {       int choice ;
            clrscr( );
            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") ;


        }
        getch( );
        }
```

Output:

    Enter your choice : 2

Second

Third

Wrong Choice

Here value of choice matches with second case so all the statements after case 2 are executed sequentially. The statements of case 3 and default are also executed in addition to the statements of case2. This is known as falling through cases.

Suppose we don't want the control to fall through the statements of all the cases under the matching case, then we can use break statement.

Break statement:

    Break statement is used inside lops and switch statements. Sometimes it becomes necessary to come out of the loop even before the loop condition becomes false. In such a situation, break statement is used to terminate the loop. This statement causes an immediate exit 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 following break 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 ;
      clrscr( );
      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 switch*/
                getch( );
        } /* End of main( ) */


Output:
        Enter your choice : 2 ↵


/* Program to perform arithmetic calculation on integers */
#include<stdio.h>
        main( )
        { char op ;
          int a, b ;
          clrscr( );
        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:
```

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

70

```
            printf ("Enter your valid operation") ;
    } /* end of switch */
    getch( );
    } /* end of main( ) */
```

```
/* Program to find whether the alphabet is a vowel or consonant */
#include<stdio.h>
    main( )
    { char ch ;
      clrscr( );
      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 */
    getch( );
    } /* end of main( ) */
```

Continue statement:

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Instead the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. The general syntax :
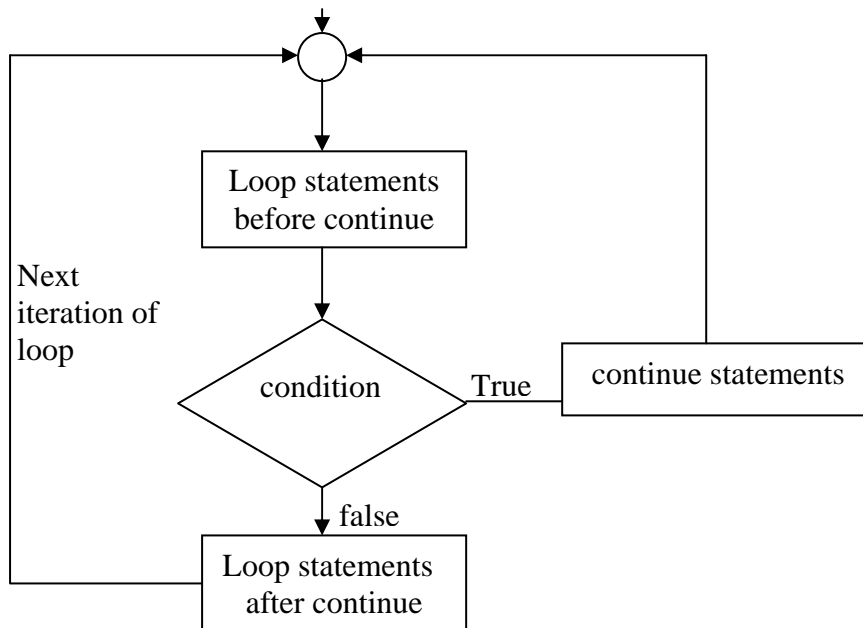
continue ;

fig : flowchart of continue control statemen

```
/* Program to demonstrate continue statement */
#include<stdio.h>
     main( )
     {
       int i, num ;
       clrscr( );
       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 */
       getch( );
     } / * end of main( ) */
```

Output:

Enter a number : 20

The even numbers from 2 to 20 are

  2 4 6 8 10 12 14 16 18 20

goto statement:

The goto statement is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program. The goto statement transfers the control to the labeled statement somewhere in the current function. The general syntax of goto statement:

```
goto label ;
    - - - - -
    - - - - -
label :
    statement ;
    - - - - -
    - - - - -
```

Here, label is any valid C identifier and it is followed by a colon. Whenever, the statement goto label, is encountered, the control is transferred to the statement that is immediately after the 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 of loop during current pass.
- Jumping completely out of the loop under certain conditions, terminating the execution of a loop.

```c
/* Program to print whether the number is even or odd */
#include<stdio.h>
    main( )
    {
    int n ;
    clrscr( );
    printf( "Enter the number :") ;
    scanf ("%d", &n) ;
    if (n%2 = = 0)
            goto even ;
            else
                    goto odd;
            even :
                    printf ("Number is even") ;
                    goto end ;
```

73

          odd :

          printf (“Number is odd”) ;

      end :

          printf (“\n”) ;

          getch( );

    }

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 statement or goto statement. In a similar way, we can jump out of a program by using the library function exit( ). In case, due to some reason, we wish to break out of a program and return to the operating system. The general syntax is

          - - - - -

          - - - - -

          if (condition) exit (0) ;

          - - - - -

          - - - - -

The exit( ) function takes an integer value as its argument. Normally zero is used to indicate normal termination and non zero value to indicate termination due to some error or abnormal 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 ;
      clrscr( );
      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 */
        getch( );
}       /* end of main() */s
```

Some Others Programs:

Q.Write a program to check whether the number is prime or not.

```
/*progam to check the num is prime or not */
/*prime.c*/
#include<stdio.h>
main( )
{
     int num,i,c;
     clrscr ( ) ;

     printf("Enter number:");
```

```
        scanf("%d",&num);
         i=2,c=0;

                while(i<=num-1)
                {
                if(num% i= =0)
                c++;
                i++;
                }
        if(c= = 0)

        printf("%d is prime number\n ",num);
        else
        printf("%d is not prime number\n",num);

        getch( );

}
```
Output:
Enter number:3
3 is prime number

```
/*write a program to calculate 2+4 +6+8 ……………………….to n terms*/
main( )
{
      int n,sum=0,i,term;
      clrscr( );
      printf("Enter how many tems :");
      scanf("%d",&n);
for(I =1, term = 2; i <= n; i++, term + = 2)
      sum +=term;
      printf("\n The is :\t %d",sum);
      getch();
                }
/*write a program to calculate 1+1/x+1/x2+1/x3……………………….to n terms*/
main( )
{
      int n,x,sum=0,i,term;
      clrscr( );
      printf("Enter how many tems :");
      scanf("%d",&n);
```

```
printf("Enter value of x:");
    scanf("%d",&x);
for(I =1 ,term = 2;i< = n ; i++,t erm + =2)
    term * = 1/x;
    sum +=term;
    printf("\n The is :\t %d",sum);
    getch();
            }
/* WAP to show
    *
    * *
    * * *
    * * * *
*/
main( )
{
    int i,j;
    clrscr( );
    for(i=1; i< = 4; i ++)
      {      for(j=1;j< = i; j++)
                    printf("*");
                        printf("\n");

                    }
                    getch( );
}
PU2009
/* WAP to show
      *
     * * *
    ** * * *
    * * * * * * *
*/

main( )
{
    int i,j,n;
    clrscr( );
    printf("Enter value of n:");
    scanf("%d",&n);
```

77

```
        for(i=1; i < = 2*(n+1- i); i++)
          {        for(j=1;j< = n-i;j++)
                        printf(" ");
                        for(j=1;j < = i;j++)
                                printf("*");
                        for(j=1;j < = i;j++)
                                printf("*");
                                printf("\n");
                        }
                        getch( );
}
/* WAP to check whether the number is palindrome or not */
  main( )
  {
      long n,num,rev=0 ;
      int  digit;
      clrscr( );
      printf ("Enter the number to be checked: \t") ;
      scanf ("%ld", &num) ;
      n=num;
  do{
        digit=num%10;
      rev=rev*10+digit;
  }while;
      if(n = = rev)
        printf (" \n The  number is palindrome" ) ;
      else
        printf (" \n The  number is not palindrome" ) ;

      getch( );
}
```

Output:

        Enter the number to be checked :   32123

        The number is palindrome

```
/* WAP to find the roots of quadratic equation */
#include<math.h>/*used for math function like sqrt() and fabs()*/
  main( )
  {
      Float a, b, c, x1,x2,d,real ,img;

      clrscr( );
```

Computer Programming in C

```c
printf ("\n Enter the values of a,b,c in ax^2+bx +c=0: \t") ;
scanf ("%f%f%f", &a , &b ,&c) ;
d = b * b- 4 * a * c;
if(d < 0)
{
   d = sqrt(fabs(d));/*saqr used for square root & fabs used for absolute value*/
   printf (" \n The  roots are imaginary :" ) ;
   real = - b/(2 * a);
   img = d/(2 * a);
   printf (" x1= % .2 f + i %f .2f ", real , img );
   printf (" x1= % .2 f + i %f .2f ", real , - img );
         }
else
   {
   d = sqrt(d);/*saqr used for square root & fabs used for absolute value*/
   printf (" \n The  roots are imaginary :" ) ;
   real = ( - b + d)/(2 * a);
   img =  (-b – d )(2 * a);
   printf (" x1 =  % .2 f \t x2 = %f .2f ", real , img );
         }
         getch( );
}
```

Some Important Questions:

1. What is a control statement? Differentiate between do while and while statement with the help of flow charts.

2. What do you mean by control statements? Describe the application of break and continue in C-programming. Explain with examples

3. Write a program to check the given number is prime or not.

4. Write a program to check number is palindrome or not.

5. What is control statement? Describe different control statement

6. Describe the different types of looping statement used in C with flow chart.

7. Write a menu driven program using switch statement having the following options.

       (i) Addition

       (ii) Subtraction"

       (iii) Multiplication

       (iv) Division

And perform the above-mentioned job as per user's choice.

Chapter – 7

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( ) is always present in each program which is executed first and other functions are optional.

7.1 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 divide and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task. So, the use of C functions modularizes and divides the work of a program.

2. When some specific code is to be used more than once and at different places in the program, the use of function avoids repetition of that code.

3. The program becomes easily understandable. It becomes simple to write the program and understand what work is done by each part of the program.

4. Functions can be stored in a library and reusability can be achieved.

7.2 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 Library and they are not required to be written by a programmer. The function's name, its return type, their argument number and types have been already defined. We can use these functions as required. 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. The user has choice to choose its name, return type, arguments and their types. The job of each user defined function is as defined by the user. A complex C program can be divided into a number of user defined functions. For example:

```
#inlcude<stdio.h>
double convert (int) ; /* function prototype */
```

```
main( )
{
        int c ;
        double d ;
        crlscr( );
        printf ("Enter temperature in Celsius: ") ;
        scanf ("%d", &c) ;
         d = convert(c) ; /*Function call*/
        printf ("The Fahrenheit temperature of %d C = %lf  F",c, d) ;
          getch( );
}
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 is defined or fixed by the language. The return type, argument and body of the function are defined by the programmer as required. This function is executed first, when the program starts execution.

7.2   Function Declaration or Prototype:

The function declaration or prototype is model or blueprint of the function. If functions are used before they are defined, then function declaration or prototype is necessary. Many programmers prefer a "top-down" approach in which main appears ahead of the programmer defined function definition. Function prototypes are usually written at the beginning of a program, ahead of any user-defined function including main(). Function prototypes provide the 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 the function.

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) ;

81

where, return_type specifies the data type of the value returned by the function. A function can return value of any data type. If there is no return value, the keyword void is used. 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 from some 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 specifier followed by an identifier like any regular variable declaration. (for eg: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces{ }.
- The first line of the function definition is known as function header.

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:

The return statement is used in a function to return a value to the calling function. It may also be used for immediate exit from the called function to the calling function without returning a value.

This statement can appear anywhere inside the body of the function. There are two ways in which it can be used:

    return ;
    return (expression) ;
    where return is a keyword. The first form of return statement is used to terminate the function without returning any value. In this case only return keyword is written.

```c
/* Program to understand the use of return statement */
#include<stdio.h>
void funct( int, float) ;
main()
{     int age ;
      float ht ;
      clrscr( );
      printf ("Enter age and height :") ;
      scanf ("%d%f", &age, &ht) ;
      funct (age, ht) ;
      getch( );
}
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)) ;

Calling a function (or A calling function):

A function can be called by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas in the main() function. The syntax of teh call if function return_type is void is: function_name (parameter name) ;

If function return int, float or any other type value then we have to assign the call to same type value like

variable = function_name(parameter) ;

For example:

m = mul(x, y)

The type of m is same as the return type of mul function. So, we can write

printf ("%d", mul(x, y)); also as printf ("%d", m) ;

Actual and Formal Parameters:

The arguments are classified into

1. Actual
2. Formal

Actual: When a function is called some parameters are written within parenthesis. These are known as actual parameter. For example

main()

{ - - - - -

convert(c) ;   actual parameter

  - - - - -

}

Formal parameters are those who are written in function header. For example

double convert (int a) /* function header */

{   formal parameter

}

Types of functions:

The functions can be classified into four categories on the basis of the arguments and return values:

1. Functions with no arguments and no return value
2. Functions with no arguments and a return value
3. functions with arguments and no return value
4. Functions with arguments and a return value

Functions with no arguments and no return value:

When a function has no arguments, it does not receive any data from the calling function. Similarly, when called function does not return a value, the calling function does not receive any data from it. Thus, in such type of functions, there is no data to transfer between the calling function and the called function. This type of function is defined as

```
void function_name( )
{
  /* body of function */
}
```

The keyword void means the function does not return any value. There is no argument within parenthesis which implies function has no argument and it doesn't receive any data from the called function.

```
/* Program to illustrate the function with no arguments and no return values */
void add( )
{
    int a, b, sum ;
    printf ("\n Enter two numbers: \t") ;
    scanf ("%d%d", &a, &b) ;
    sum = a + b;
    printf ("\n The sum is %d", sum) ;
}
    void main( )
    {
      clrscr( );
      add( ) ;
      getch( );
    }
```

Output:

Enter two numbers: 3   4 ↵
The sum is 7

Functions with no arguments and a return value:

These type of functions do not receive any arguments but they can return a value.

```
int func(void)
main()
{
    int r ;
    - - - -
}
```

```
    int func(void)
{
    - - - -
    - - - -
    return (expressions) ;
}
```

/* Program to illustrate function with no argument and a return value */
```
#inlcude<stdio.h>
    int add(void) ;
    main )
    {
      printf (" The sum is %d\n", add( ) );
    }
      int add(void)
    {
      int n1, n2, sum ;
      printf ("Enter two numbers: \t") ;
      scanf ("%d%d", &n1, &n2) ;
      sum = n1+n2 ;
      return (sum) ;
    }
```
Output:
    Enter two numbers: 3 4 ↵
    The sum is 7

Functions with arguments and no return values:
    These types of functions have arguments, hence the calling function can send data to the
called function but the called function does not return any value. These functions can be
written as
```
void func (int, int) ;
main( )
{ - - - -
  func(a, b) ;
  - - - -
}
void func( int c, int d)
{
  - - - -
  statements
  -   - - -
```

}

/* Program to illustrate the functions with arguments but no return values */
void add(int, int) ;
main( )
{
     int a, b
     printf ("Enter two numbers: \t") ;
     scanf ("%d%d", &a, &b) ;
     add (a, b)
}
void add(int c, int d)
{
     int sum ;
     sum = c+d;
     printf (" \n The sum is %d", sum) ;
}
Output:
     Enter two numbers: 3 4 ↵
     The sum is 7

Functions with arguments and return values
     These types of functions have arguments, so the calling function can send data to the called function, it can also return any values to the calling function using return statement. This function can be written as
     int func (int, int) ;
     main( )
     { int r ;
       - - - - -
       r = func(a,b) ;
       - - - - -
     }
     int func(int a, int b)
     {
       - - - - -
       - - - - -
       return(expression) ;
     }

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

```
int add(int, int) ;
main( )
{
  int a, b, sum ;
  clrscr();
  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

7.3   Call by value & Call by reference (Or Pass by value & Pass by reference)

Passing arguments to functions:

An argument is a data passed from a program to the function. In function, we can pass a variable by two ways.

1. Pass by value (or call by value)
2. Pass by reference (or call by reference)

Function call by value:

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameters are not changed. For eg:

```
#incldue<stdio.h>
void swap(int, int) ; /*function prototype*/
void main( )
{
    int x, y ;
    clrscr( ) ;
    x = 10 ;
    swap (x,y) ; /*function call by value */
    printf ("x = %d \n", x) ;
```

```
        printf ("y=%d\n", y)
        getch( ) ;
}
void swap (int a, int b)  /*function definition */
{
        int t ;
        t = a ;
        a = b ;
        b = t ;
}
Output x = 10
        y = 20
```

Function call by reference:

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

Pointer:

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 always preceded /xy '*' (asterik) operator. Thus pointer variables are defined as

```
        int *a ;
        float *b ;
        char *c ;
```

where a, b, c are pointer variable which stores address of integer, float and char variable. Thus the following assignments are valid.

```
        int x ;
        float y ;
        char c ;
        a = &x ; /* the address of x is assigned to pointer variable a */
        b = &y ; /* the address of y is stored to pointer variable b */
        c = &c ; /* the address of c is stored to pointer variable c */
```

```
#include<stdio.h>
void swap(int *, int * ) ; /* function prototype */
main ( )
 {
        int x, y ;
        x = 10 ;
        y = 20 ;
```

```
clrscr( );
swap (&x, &y) ; /* function call by address */
printf ("x=%d\n", x) ;
printf ("y=%d\n", y) ;
getch( ) ;
}
void swap(int *a, int *b)
{
  int t ;
  t = *a ;
  *a = *b ;
  *b = t ;
}
```

Output:

     x = 20

     y=10

[P.U.2005]

7.5   Concept of Local, Global & Static variables:

Local variables (automatic or internal variable) :

      The variables that are defined within the body of a function or a block, are local to that function or block only and are called local variables. The local variables are created when the function is called and destroyed automatically when the function is exited. The keyword auto may be used storage class specification while declaration of variable. A variable declared inside a function without storage class specification auto is, by default, an automatic variable.

      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.

      Storage class refers to the performance of a variable and its scope within the program.

Initial Value: This is the initial value assigned by the language to a variable if no value is assigned to variable explicitly by the programmer.

Scope: Scope of a variable can be defined as the region over which the variable is visible or valid.

Life time: The period of time during which memory is associated with a variable is called the life time or extent of the variable.

```
/* Program to illustrate local variable */
long int fact (int n)
{
      int i ;
      long int f = 1 ;
      for (i=1 ; i<=n; i++)
          f * = i        ;
      return ( f ) ;
{
main( )
{
      int num ;
clrscr( );
      printf ("Enter a number :") ;
      scamf (%d", &num) ;
      printf ("The factorial of %d is %ld", num, fact(num));
}
Output:
      Enter a number : 5 ↵
      The factorial of 5 is 120
```

Global Variables (External):

The variables that are defined outside any function are called global variables. All functions in the program can access and modify global variables. IT is useful to declare a variable global if it is to be used by many functions in the program. The default initial value for these variable is zero. The scope is global i.e. within the program. The life time is as long as the program's execution does not come to an end.

An external variable must begin with the storage class specifier extern. A variable declared outside a function without storage class specification extern is, by default, an external variable but defined after some function.

```
/* A program to illustrate the global variables */
int a = 10 ;
void  func( )
{
      a = 20
      printf ("\t%d",a++) ;
}
```

91

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

Output : 10   20   21

<u>Illustratration of extern variable:</u>

```
main( )
{
     extern float marks ;
      - - - - -
}
func1( )
{
     extern float marks ;
     - - - - -  ;
}
     float marks ; /* global space but defined after function */
```

The extern declaration does not allocate storage space variables. The extern declaration of marks inside the function informs the compiler that marks is a float type extern variable defined somewhere else in the program.

<u>Static variables :</u>

Static variables are declared by writing keyword static in front of the declaration.

static type var_name ;

A static variable is initialized once and the value of a static variable is retained between function call. If a static variable is not initialized then it is automatically initialized to O. 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.

```
/* Program with auto variable */        /* Program with static variable */
void increment( )                       void increment( )
{                                       {
    int i = 1                               static int i = 1
    printf ("%d\n", i ) ;                   printf ("%d\n", i ) ;
    i++ ;                                   i++ ;
}                                       }
```

```
    main( )                                  main( )
{   clrscr( );                           {
    increment( ) ;                           increment( ) ;
    increment() ;                            increment( ) ;
    increment() ;                            increment( ) ;
    increment() ;                            increment( ) ;
    getch( );                                getch( );
    }                                        }
```

Output :

1	1
1	2
2	3
1	4

Register variable:

     Register variables are special case of automatic variables. Automatic variables are allocated storage in the memory of the computer; however, for most computers accessing data in memory is considerably slower than processing in the CPU. These computers often have small amounts of storage within the CPU itself where data can be stored and accessed quickly. These storage cells are called registers. If a variable is declared as register variable, then it is stored in the CPU register. They are allocated storage upon entry to a block ; and the storage is freed when the block is exited. The scope of register variables is local to the block in which they are declared. Rules for initializations for register variables are the same as for automatic variables. For eg.

```
    main( )
    {
      register int a = 10 ;
      - - - - -
      - - - - -
    }
```

7.6   Recursive Function:

     Recursion in programming is a technique for defining a problem in terms of one or more smaller versions of the same problem. The solution of the problem is built on the results from the smaller versions. A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call. Thus, the function is called recursive function if it calls to itself and recursion is a process by which a function call itself repeatedly until some specified condition will be 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.

93

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

- Problem could be written or defined in terms of its previous result.
- Problem statement must include a stopping condition i.e. we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise function will never return.

```
/* WAP to find the factorial of a number using recursive method */
long int factorial (int n)
{
    if (n = = 1)
        return (1) ;
    else
        return (n*factorial(n-1)) ;
}
main( )
{   int num ;
    printf ("Enter a number : ") ;
    scanf ("%d", &num) ;
    printf ("The factorial is %ld", factorial (num)) ;
}
```

Output:

Enter a number : 5 ↵

The factorial is 120

Recursion	Iteration
1. A function is called from the definition of the same function to do repeated.	1. Loop is used to do repeated task.
2. Recursive is a top-down approach to problem solving.	2. Iteration is like bottom-up approach.
3. In recursion, a function calls to itself until some condition will be satisfied.	3. In iteration, a function doesn't call to itself.
4. Problem could be defined in terms of its previous result to solve a problem using recursion.	4. It is not necessary to define a problem in term of its previous result to solve using iteration.
5. All problems cannot be solved using recursion.	5. All problems can be solved using iteration.

The Towers of Hanoi:

The Towers of Hanoi is a well known children's game played with three poles and a number of different size disk. Each disk has a hole in the centre, allowing it be stacked

around any of the poles. Initially, the disks are stacked on the leftmost pole in the order of decreasing size i.e. the largest on the bottom and the smallest on the top.

| left | Center | Right |
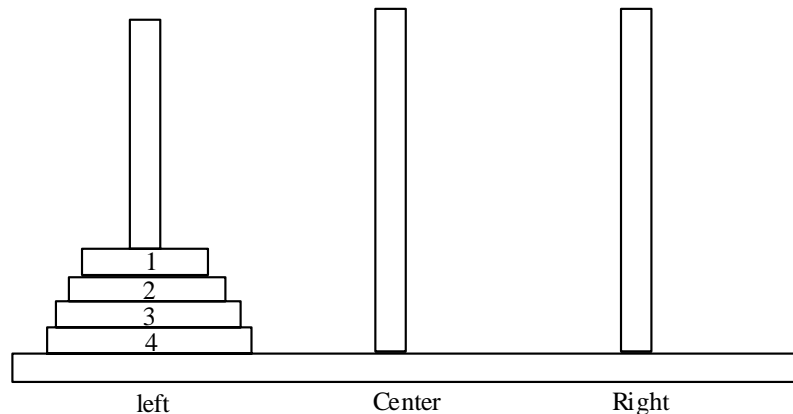
The object of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on the top of a smaller disk. Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for immediate storage. Thus allowing the disks to be moved without placing a larger disk over a smaller one. Assume there are n disks, numbered from smallest to largest. If the disks are initially stacked on the left pole, the problem of moving all n disks to the right pole can be stated in the following recursive manner.

1. Move the top n-1 disks from the left pole to the centre pole.
2. Move the nth disk (the largest disk) to the right pole.
3. Move the n-1 disks on the centre pole to the right pole.

The problem can be solved in this manner for any value of n greater than, 0(n=0 represents a stopping condition). In order to program this game, we first label the poles so that the left pole is represented as L, the centre pole as C and the right pole as R. We then construct a recursive function called transfer that will transfer n disks from one pole to another. Let us refer to the individual poles with the char type variable from, to and temp for the origin, destination, and the temporary storage respectively. Thus, if we assign the character L to from, R to and C to temp, we will in effect be specifying the movement of n disks from the leftmost pole to righmost pole, using the centre pole for immediate storage.

```
/* The Tower of Hanoi_solved using recursion */
#include<stdio.h>
void transfer (int n, char from, char to, char temp) /*function prototype */
main( )
{
```

```
        int n ;
        printf ("Welcome to the Tower of Hanoi \n\n") ;
        printf ("How many disk?") ;
        scanf ("%d", &n) ;
        printf ("\n") ;
        transfer (n, 'l', 'R', 'C') ;
}
void transfer (int n, char from, char to, char temp)
/* transfer n disks from one pole to another */
/*      n = number of disks
        from = origin
        to = destination
        temp = temporary storage */
{
        if(n>0) {
        /* move n-1 disk from origin to temporary */
          transfer (n-1, from, temp to) ;
        /* move nth disk from origin to destination */
        printf ("Move disk %dfrom%cto%c\n", n, from, to) ;
        /* move n-1 disks from temporary to destination */
          transfer (n-1, temp, to, from) ;
                    }
        return ;
}


/* Program : Fibonacci number by recursion */
#include<stdio.h>
#include<conio.h>
        int fib(int x)
{
        if (x = = 0  || x = =1)
        return 1 ;
         else
           return(fib(x-1) + fib(x-2)) ;
}
main()
{
        int i, n, no ;
        clrscr( ) ;
        printf ("how many no in series : ") ;
        scanf("%d", &n) ;
```

```
     for (i = 1 ; i<=n ; i++)
       {
               no = fib(i) ;
               printf("%d", no) ;
       }
     getch( ) ;
}


  /* WAP to add the natural numbers using recursive method */
  long int add (int n)
  {
     If(n = =0)
     return  0;
     else
  return (n + add(n-1)) ;
  }
  main( )
  {
     int num ;
     clrscr( );
     printf ("Enter How many  numbers : ") ;
     scanf ("%d", &num) ;
     printf ("The sum of natural number is %ld", add(num)) ;
     getch( );
}
```

Output:

```
     Enter How  many numbers :5
     The sum of natural numbers is 15
```

More about main( ) :

We know that every C program should have one main( ) function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact main can take two arguments called argi and argu and the information contained in the command line (i.e. C>PROGRAM X_FILE Y_FILE) is passed on to the program through these arguments when main( ) is called up by the system.

The variable argc is an argument counter that the number of arguments on the command line. The argu is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of argc. For

97

instance, for the command line given above, argc is three and argu is an array of three pointers to strings as shown below:

argu[0]   PROGRAM

argu[1]   X_FILE

argu[2]   Y _FILE

In order to access the command line arguments, we must declare the main function and its parameters as follows:

main(argc, argu) ;

int argc ;

char *argu[] ;

{

{

WAP that will receive a filename and a line of text as command line arguments and write the text to the file.

The command line

F12_6   TEXT   AAAAAA   BBBBBB - - - - - - - GGGGGG

Each word in the command line is an argument to the main and therefore the total number of argument is 9.

The argument vector argv[1] points to the string TEXT and therefore the statement.

fp = fopen(argv[1], "w") ; opens a file with name TEXT. The for loop that follows immediately write the remaining 7 arguments the file TEXT.

Program also prints two output, one from the file TEXT and the other from the system memory. The argument vector argv contains the entire command line in the memory and therefore the statement:

printf ("%s\n", i*s, argv[i]) ;

prints the argument from the memory.


```
#include<stdio.h>
    main(argc, argv)          /* main with arguments */
    int argc ;                /* argument count */
    char * argv[ ] ;          /* list of arguments */
{
    file *fp ;
    int i ;
    char word[15] ;
    fp=fopen(argv[1], "w") ;          /* open file with name arg[1] */
    printf("/n No arguments in command line = %d\n\n", argc) ;
    for (i = 2 ; i<large ; i++)
{
```

```
    fscanf (fp, "%s", word) ;
    printf("%s", word) ;
}
    fclose(fp) ;
    printf (\n\n") ;
/* Writing the arguments from memory */
    for (i = 0; i<argc ; i++)
    printf("%*s\n", i*5, argv[i] ;
}
```

Output:

```
    C>F12_6        TEXT        AAAAAA - - - - - - GGGGGG
    No of arguments in command line = 9
    contents of TEXT file
    AAAAAA        BBBBBB - - - - - GGGGGG
    C:\C\F12_6.EXE
    TEXT
      AAAAAA
            BBBBBB
                CCCCCC
                    DDDDDD
                        EEEEEE
                            FFFFFF
                                GGGGGG
```

Some Important Questions:

1.Define function and list all the types of user defined function used in C. How do you differentiate library function from user defined function?

2.Write a program to calculate factorial of 'n' number by using recursive function where n is the number inputted by user.

3.What are local and global variables? Determine what would   be the output of the following:

```
        int i=l;
        printf("i=%d i=%d i=%d",i,  i + +, ++i)
```

4.What is the method of declaring local and global variables?

5.What is a recursive function? Write a program in C to allow a user to enter an integer number interactively and display its factorial value.

6.What is a static variable? Write a programin C that uses a static variable.

7.What is a function? Give the syntax of declaring a function in C.

8.Explain how an array can be passed to a function with the help of a suitable program in C.

9.What is Differences between call by value and call by reference.

99

Chapter – 8

Arrays and Strings

8.1 Introduction:

An array is a collection of same type of data item which are stored in consecutive memory locations under a common name. Suppose, we have 20 numbers of type integer and we have to sort them in ascending or descending order. If we have no array, we have to define 20 different variable like a1, a2, ..., a20 of type int to store these twenty numbers which will be possible but inefficient, If the number of integers increase the number of variables will also be increased and defining different variables for different numbers will be impossible and inefficient. In such situation where we have multiple data items of same type to be stored, we can use array. In array system, an array represents multiple data items but they share same name. The individual elements are characterized by array name followed by one or more subscripts or indices enclosed in square brackets. 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 (i.e. auto, register, static or extern)

8.2 Single and Multi-dimension arrays:

One or single dimensional array:

There are several forms of an array in C-one dimensional and multi-dimensional array. In one dimensional array, there is a single subscript or index whose value refers to the individual array element which ranges form 0 to n-1 where n is the size of the array. For e.g. int a[5] ; is a declaration of one dimensional array of type int. Its elements can be illustrated as

1st element	2nd	3rd	4th	5th element
a[0]	a[1]	a[2]	a[3]	a[4]
2000	2002	2004	2006	2008

The elements of an integer array a[5] are stored 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. The general syntax of array is [i.e. declaration of an array :]

data_type array_name[size];

or if we want to add storage classes then that look like:

storage_class data_type array_name[size] ;

where

- storage_class refers to the storage class of the array. It may be auto, static, extern and register, It is optional.
- data_type is the data type of array. It may be int, float, char, etc.

- array_name is name of the array. It is user defined name for array. The name of array may be any valid identifier.
- size of the array is the number of elements in the array. The size is mentioned within[ ]. The size must be an int constant like 10, 15, 100, etc or a constant expression like symbolic constant. For example

#define size 80

a [size] ;

The size of the array must be specified [i.e. should not be blank) except in array initialization.

int num[5] ; i.e. num is an integer array of size 5 and store 5 integer values

char name[10] ; i.e. name is a char array of size 10 and it can store 10 characters.

float salary[50] ; i.e. salary is a float type array of size 50 and it can store 50 fractional numbers.

Initialization of array:

The array is initialized like follow if we need time of declaration

data_type array_name[size] = {value1, value2, ..., valuen} ;

For eg.

1. int subject[5] = {85, 96, 40, 80, 75} ;
2. char sex[2] = {'M', 'F'} ;
3. float marks[3] = {80.5, 7.0, 50.8} ;
4. int element[5] = {4, 5, 6}

In example(4), elements are five but we are assigning only three values. In this case the value to each element is assigned like following

element[0] = 4

element[1] = 5

element[2] = 6

element[3] = 0

element[4] = 0

i.e. missing element will be set to zero

Accessing elements of array:

If we have created an array, the next thing is how we can access (read or write) the individual elements of an array. The accessing function for array is

array_name[index or subscript]

For e.g.

int a[5], b[5] ;

a[0] = 30 ; /* acceptable */

```
b = a ; /* not acceptable */
if (a<b)
{_ _ _ _ } /* not acceptable */
```
we can assign integer values to these individual element directly:
```
a[0] = 30 ;
a[1] = 31 ;
a[2] = 32 ;
a[3] = 33 ;
a[4] = 34 ;
```

A loop can be used to input and output of the elements of array.

Examples of single dimensional array:
```
/* 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 output */
     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 */
    } /* end of outer loop */
    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
```

Multi-dimensional arrays:

An array of arrays is called multi-dimensional array. For example a one dimensional array of one dimensional array is called two dimensional array.

A one dimensional array of two dimensional arrays is called three dimensional arrays, etc. The two dimensional array are very useful for matrix operations.

Declaration of two dimensional array:

Multidimensional arrays are declared in the same manner as one dimensional array except that a separate pair of square brackets are required for each subscript i.e. two dimensional array requires two pair of square bracket ; three dimensional array requires three pairs of square brackets, four dimensional array require four pair of square brackets, etc.

The general format for declaring multidimensional array is

data_type array_name [size1] [size2] ...... [sizen] ;

We can add storage class in above declaration if necessary like follow:

storage_class data_type array_name[size1] [size2] ...... [sizen] ;

where storage_class part is optional. For example:

1. int n[5] [6] ;
2. float a[6] [7] [8] ;
3. static char line [5] [6] ;
4. double add [6] [7] [8] [9] ;

Initialization of multidimensional array:

Similar to one dimensional array, multidimensional array can also be initialized. For e.g.

1. int [3] [2] = {1, 2, 3, 4, 5, 6) ;

The value is assigned like follow:

a[0] [0] = 1 ;
a[0] [1] = 2 ;
a[1] [0] = 3 ;
a[1] [1] = 4 ;
a[2] [0] = 5 ;
a[2] [1] = 6 ;

2. int a[3] [2] = { {1,2}, {3,4}, {5,6} } ;

In above example, the two value in the first inner pair of braces are assigned to the array element in the first row, the values in the second pair of braces are assigned to the array element in the second row and so on. i.e.

a[0] [0] = 1 ;   a[0] [1] = 2 ;
a[1] [0] = 3 ;   a[1] [1] = 4 ;
a[2] [0] = 5 ;   a[2] [1] = 6 ;

3. int a[3] [3] = { {1,2}, {4,5}, {7, 8, 9} } ;

In above example, the values are less than the elements of array. The value assign like follow:

a[0] [0] = 1   a[0] [1] = 2   a[0] [2] = 0
a[1] [0] = 4   a[1] [1] = 5   a[1] [2] = 0

a[2] [0] = 7   a[2] [1] = 8   a[2] [] = 9

In above example the value zero is assigned to a[0] [2] and a[1] [2] because no value assigned to these.

4. int a[ ] [3] = {12, 34, 23, 45, 56, 45} ; is perfectly acceptable.

It is important to remember that while initializing a 2-dimensional array, it is necessary to mention the second (column) size where as first size (row) is optional.

In above example, value assign like follow

a[0] [0] = 12   a[0] [1] = 34   a[0] [2] = 23
a[1] [0] = 45   a[1] [1] = 56   a[1] [2] = 45

int a[2] [ ] = {12, 34, 23, 45, 56, 45} ;
int a[ ] [ ] = {12, 34, 23, 45, 56, 45} ;

would never work.

Accessing elements of multidimensional array:

We can access the multidimensional array with the help of following accessing function:

array_name [index1] [index2]. . . . . . . [indexn]

For example:

int a[5] [6] ;

We can write the following statement

1. a[0] [2] = 5 ;
2. x = a [3] [2] ;
3. printf("%d", a[0] [0]) ;
4. printf("%d", a[2] [2]) ;

If we want to read all the elements of above array, we can make a loop like

for (i=0; i<=4; i++)
for (j=0; j<=5; j++)
        scanf("%d", &a[i] [j]) ;

If we want to print all the elements of above array a then the loop is

for (i=0;j<=4;i++)
for (j=0; j<=5; j++)
        printf("%d", a[i] [j]) ;

8.3 Processing an array:

```
/* Program to read & display 2×3 matrix */
#include<stdio.h>
#include<conio.h>
void main( )
{
    int matrix [2] [3], i, j ;
```

105

```
        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] :     1
        Enter matrix [0] [1] :     2
        Enter matrix [0] [2] :     3
        Enter matrix [1] [0] :     4
        Enter matrix [1] [1] :     5
        Enter matrix [1] [2] :     6
        Enter 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]) ;
          { 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 ;
      clrscr( );
        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) ;
```

107

```
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

#Solve the Matrix

$$a = \begin{pmatrix} 2 \\ 2 \end{pmatrix}_{2\times1} \qquad\qquad b = \begin{pmatrix} 3 & 3 \end{pmatrix}_{1\times2}$$

$$s = \begin{pmatrix} 6 & 6 \\ 6 & 6 \end{pmatrix}_{2\times2}$$

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}_{2\times2} \qquad\qquad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}_{2\times2}$$

$$A\times B = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cg+dh) \end{pmatrix}_{2\times2}$$

8.3   Passing arrays to functions:

Like any other variables, we can also pass entire array to a function. An array name can be named as an argument for the prototype declaration and in function header. When we call the function no need to subscript or square brackets. When we pass array that pass as a call by 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 */
```

109

```
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 then
calculate 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);
```

111

```
        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
```

8.4Arrays of Strings:

String:

Array of character type which is terminated by null characters is known as string. In other words, we can say that a string is a sequence of contiguous character in memory terminated by the null character '\0'. The terminating null character '\0' is important because it is the only way the string handling functions can know there the string ends. Normally, each character is stored in one byte, and successive characters of the string are stored in successive bytes. In C, header file string.h provides special function for manipulating strings.

Initializing string:

A string is initialized as

char name[ ] = { 'A', 'R', 'U', 'N' }

Then string name is initialized to ARUN. This technique is also valid. But C offers special way to initialize the string as

char name[ ] = "Arun" ;

The characters of the string are enclosed within a pair of double quotes.

```
/* Program to illustrate string initialization */
#include<stdio.h>
#include<conio.h>
void main( )
{      char name[ ] = "ARUN KUMAR" ;
       clrscr( ) ;
       printf ("Your name is :%s\n", name)
       getch( ) ;
{
```
Output:
Your name is:
ARUN KUMAR

Arrays of strings:

Arrays of string means two dimensional array of characters. For example

char str[5] [8] ;

The first dimension (size) tells how many strings are in the array. The second dimension tells the maximum length of each string. In above declaration, we can store 5 strings, each can store maximum 7 characters ; last : 8th space is for null terminator in each string.

Initialization of array of string:

The array of string initialized like follow:

(1) char str[2] [7] = {"Sunday", "Monday") ;

or like the following

char str[2] [7] = { {'s', 'u', 'n', 'd', 'a', 'y'} ;
    {'m', 'o', 'n', 'd', 'a', 'y'} } ;

Accessing or array of string:
    char a[4] [7] ;
If we want to read all the string from keyboard, we can use loop like follow:
for (i=0 ; i<=3 ; i++)
scanf("%d", a[i]) ;
If we want to print all the string then loop is
for (i=0 ; i<=3 ; i++)
printf ("%s", a[i]) ;


```
/*Program to store name of week days and then print all */
#inclue<stdio.h>
#inclue<conio.h>
void main( )
{
    int i ;
    char  day[7] [10] = {"sunday", "monday", "tuesday", "wednesday", "thursday",
"friday", "saturday" } ;
clrscr( ) ;
printf("Weekdays are: \n") ;
for (i=0 ; i<=6 ; i++)
printf ("%s\n", day[i]) ;
getch() ;
3
```
Output:
    Weekdays are
      Sunday
      Monday
      Tuesday
      Wednesday
    Thursday
    Friday
    Saturday

8.6 String Handling Function:

The library or built-in functions strlen( ), strcpy( ), strcat( ), strcmp( ), strrev( ) etc are used for string manipulation. In order to use these function, we must include string.h file.

Computer Programming in C

1. <u>strlen( ):</u>

This function returns an integer which denotes the length of string passed. The length of string is the number of characters present in it, excluding the terminating null character. Its syntax is

integer_variable = strlen(string) ;

```
/* Program to find out the length of a string */
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
     char name[40] ;
     int len ;
     clrscr ;
     printf ("Enter your name : \t") ;
     gets(name) ;
     len = strlen(name) ;
     printf ("\n The number of character in your name is :\t%d", len) ;
getch( ) ;
}
```

Output:

Enter Your Name : ARUN KUMAR YADAV

The number of character in your name is : 16

2. <u>strcpy( ):</u>

The strcpy( ) function copies one string to another. The function accepts two string as parameters and copies the second string character by character into the first one upto including the null character of the second string. The syntax is

strcpy(destination_string, source_string) ;

i.e. strcpy(s1, s2) ; means the content of s2 is copied to s1.

```
/* Program to copy one string to another */
#indlue<stdio.h>
#include<string.h>
main( )
{
     char name[ ] = "Arun Kr. Yadav", s[20] ;
     strcpy (s,name) ;
     printf ("The copied string is %s\t", s) ;
}
```

115

Output:
The copied string is Arun Kr. Yadav

3.strcat( ):

      This function concatenates two strings i.e. it appends one string at the end of another. This function accepts two strings as parameters and stores the contents of the second string at the end of the first. Its syntax is:

      strcat (string1, string2) ;
      i.e. string1 = string1 + string2 ;

```
/* Program to concate two strings */
#include<stdio.h>
#include<string.h>
main( )
{
    char fname[20] = "Arun", lname[ ] = "Kumar" ;
    strcat (fname, lname) ;
    printf ("\n The full name is %s", frame) ;
}
```
Output:
      The full name is Arun Kumar

4.strump( ):

      This function compares two strings to find out whether they are same or different. This function is useful for constructing and searching strings as arranged into a dictionary. This function accepts two string as parameters and returns an integer whose value is

    1) less than 0 if the first string is less than the second
    2) equal to 0 if both are same
    3) greater than 0 if the first string is greater than the second

      Two strings are compared character by character until there is a mismatch or end of one of strings is reached. Whenever two characters in two strings differ, the string which has the character with a higher ASCII value is greater. For example, consider two string "ram" and "rajesh". The first two character are same but third character in string ram and that is in rajesh are different. Since ASCII value of character m in string ram is greater than that of j in string rajesh, the string ram is greater than rajesh. Its syntax

      integer_variable = strump(string1, string2) ;

```
/* Program to illustrate the use of strcmp( ) */
#include<stdio.h>
```

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

116

Computer Programming in C

```c
#include<string.h>
main( )
{     char name1[15], name2[15] ;
      int diff ;
      printf ("\n Enter first string \t") ;
      gets(name1) ;
      printf ("\n Enter second string \t") ;
      gets(name2) ;
      diff = strcmp(name1, name2) ;
      if (diff>0)
        printf ("%s is greater than %s by value %d", name1, name2, diff) ;
      else
        print ("%s is same as %s", name1, name2) ;
}
```
Output:
     Enter first string   ram
     Enter second string   rajesh
     ram is greater than rajesh by value 3.

5.   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". Its syntax is
     strrev(string) ;
For example:
     strrev(s) means it reverses the characters in string s and stores reversed string in s.


```c
/* Program to illustrate strrev( ) */
#include<string.h>
main( )
{
      char name[15] = "Manju Shree") ;
      char name2[15] ;
      strcpy (name2, name) ;
      printf ("The reversed string of  original string %s is %s", name2, name) ;
      3
```
Output:
     The reversed string of original string Manju Shree is eerhS   ujnaM.


```c
/* A program to read a string and check for palindrome */
void main( )
{
```

          117

```
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( ) ;
}
```

Some Important Questions:

1.What is an array? What are the advantages of using array?

2.Write a program to convert a lowercase character string into uppercase using array.

3. Write a program to sum the diagonal element of a square matrix of order n.

4.Write a program that read a 3*3 matrix and find sum of all the elements of matrix then print the sum.

5. Write short note on :String handling functions

6.Describe any three types of string handling function with example

---

Chapter – 9

Pointers

9.0 Pointers:

Before studying pointers, it is important to understand how memory is organized in a computer. The memory in a computer is made up of bytes arranged in a sequential manner. Each byte has an index number which is called address of that byte. The address of these bytes start from zero and the address of last byte is one less than the size of memory. Suppose we have 64MB of RAM (Random Access Memory), then memory will consist of 64×2^{20} = 67108864. the address of these bytes will be from 0 to 67108863.

Memory Block

Memory Address
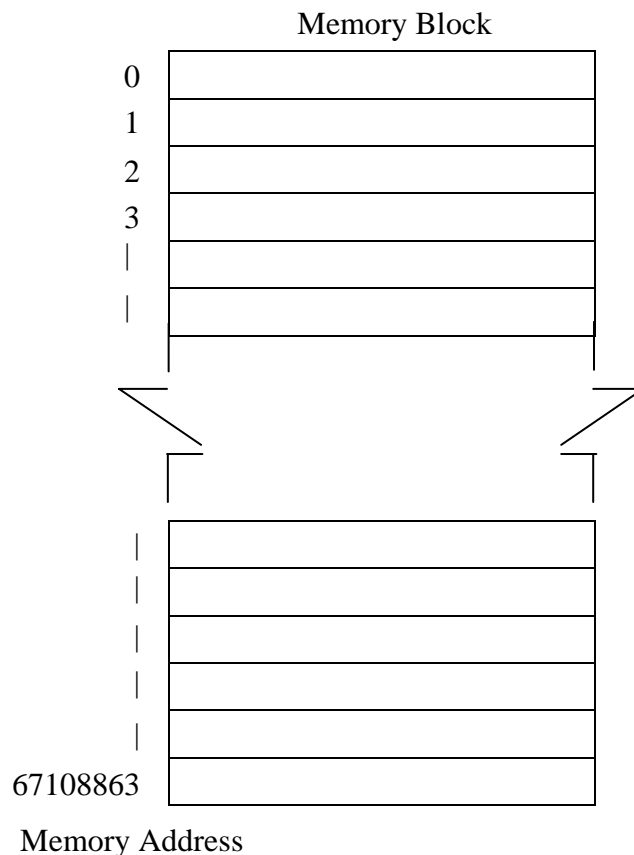
We have studied that it is necessary to declare a variable before using it, since compiler has to reserve space for it. The data type of the variable also has to be mentioned so that the compiler knows how much space need to be reserved. For example:

    int age;

The compiler reserves 2 consecutive bytes from memory for this variable and associates the name age with it. The address of first byte from the two allocated bytes is k\a the address of variable age.

Suppose compiler has reserved bytes numbered 65524 and 65525 for the storage of variable age, then the address of variable age will e 65524. Let us assign some value to this variable.

119

age = 20;

Now this value will be stored in these 2 bytes in form of binary representation. The number of bytes allocated will depend on the data type of variable. For example, 4 bytes would have been allocated for a float variable, and the address of first byte would be called the address of variable.

Address operator (&):

C provides an address operator '&', which returns the address of a variable when placed before it. This operator can be read as "the address of", so &sn means address of sn, similarly &price means address of price. The following program prints the address of variabvles using address operator.

```
/* Program to print address of variable using & */
#include<stdio.h>
 main( )
    {
      int sn          =      30;
      float price     =      150.50;
    printf("value of sn=%d, Address of sn=%u\n", sn, &sn);
    printf("value of price=%f, Address of price=%u\n", price, &price);
    }
output:
      value of sn = 30,                Address of sn    = 65524
      value of price = 150.500000     Address of price = 65520
```

9.1 Pointer Fundament:
9.2 Pointer Declaration

Introduction:

A pointer is a variable that contains a memory address of data or another variable. [In other word, a pointer is a variable that stores memory address]. Like all other variables it also has a name to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location.

Declaration and assigning (initializing) of pointer:

Like any other variable, pointer variable should also be declared before being used. The general syntax is
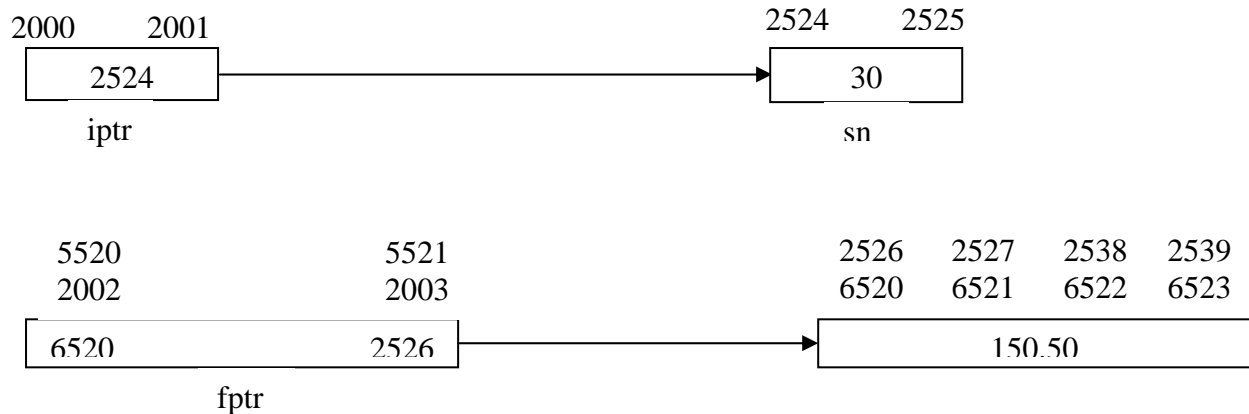
    data_type   *pname;

where pname is the name of pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as a pointer. data_type is basic data type. For example:

```
int     *iptr,sn=30;
float   *fptr,price=150.50;
iptr=&sn;                    \*assigning of pointer*\
fptr=&price;
```

Now, iptr contains the address of variable sn i.e. it points to variable sn, similarly fptr points to variable sn, similarly fptr points variable price.

2000	2001				2524	2525

2524 → 30

iptr   sn

5520	5521		2526	2527	2538	2539
2002	2003		6520	6521	6522	6523

6520   2526 → 150.50

fptr

Pointer are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain address only. Generally 2 bytes are used to store an address (may vary in different computers), so the compiler allocates 2 bytes for a pointer variable.
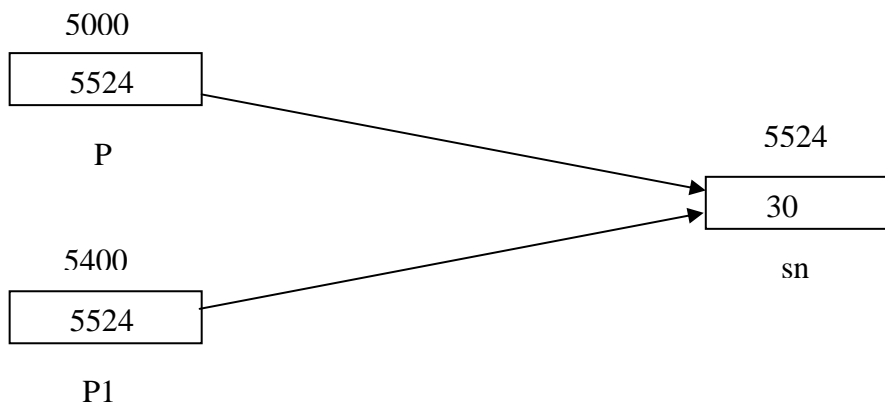
If pointers are declare after the variable like

```
int sn=30,*p=sn;
float price=150.50,*q=&price;
```

It is also possible to assign the value of one pointer variable to the other provided their base type is some.

```
P1=P;
```

Now, both pointer variable P and P1 contains the address of variable sn and points the same variable

5000

5524

P

5524

30

sn

5400

5524

P1

We can also assign constant zero to a pointer of any type. A symbolic constant NULL is defined in stdio.h which denotes the value zero. The assignment of NULL to a pointer guarantees that it does not point to any valid memory locations. This can be done as

  ptr=NULL;

<u>Application of pointer:</u>
  Some uses of pointers are
1) Accessing array elements
2) Returning more than one value from a function
3) Accessing dynamically allocated memory
4) Implementing data structure like linked lists, trees, and graphs.
5) Increasing the execution speed as they refer address.

<u>Indirection or Deference Operator:</u>
  The operator '*', used in front of a variable, is called pointer or indirection or deference operator. Normal variable provides direct access to their own values where as 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 deference. When the pointer is declared, the star indicates that it is a pointer, not a normal variable. When the pointer is deferenced, the indirection operator indicates the value at that memory location stored in the pointer. Let us take an example:

  int a = 87;
  float b = 4.5;
  int*P1 = &a;
  float*p2 = &b;

  In above program, if we place '*' before P1 when we can access the variable whose address is stored in P1. Since P1 contains the address of variable a, we can access the variable a by waiting *P1. Similarly we can access variable b by writing *P2. So we can use 8P1 & *P2 in place of variable names a and b anywhere in our program. Let us see some other examples:

  *P1=9; is equivalent to a=9;
  (*P1)++; is equivalent to a++;
  x=*P2+10; is equivalent to x=b+10;
  printf("%df",*P1,*P2); is equivalent to printf("%d%f",a,b);
  scanf("%d%f",P1,P2); is equivalent to scanf("%d%f",&a,&b);

```
#include<stdio.h>
 main( )
     { int a=50;
     float b=4.5;
     int *P1=&a;
     float *P2=&b;
```

```
        printf("value of P1 = address of a = %u\n",P1);
        printf("value of P2 = address of b = %u\n", P2);
        printf("Address of P1 = %u\n",&P1);
        printf("Address of P2 = %u\n', &P2);
        printf("value of a= %d%d%d\n",a,*P1,*(&a));
        printf("value of b = %f%f%f\n",b,*P2,*(&b);
    }
```

OUTPUT:

Value of P1 = Address of a = 65524

Value of P2 = Address of b = 65520

Address of P1 = 65518

Address of P2 = 65516

Value of a = 50   50   50

Value of b = 4.500000 4.500000   4.500000

9.3 Passing pointers to functions:

A pointer can be passed to a function as an argument. Passing a pointer means passing address of a variable instead of value of the variable. As address is passed in this case, this mechanism is also k\a call by address or call by reference. When pointer is passed to a function, while function calling, the formal argument of the function must be compatible with the passing pointer i.e. if integer pointer is being passed, the formal argument in function must be pointer of the type integer and so on. As address of variable is passed in this mechanism, if value in the passed address is changed within function, the value of actual variable also changed.

```
/* Program to illustrate the use of passing pointer to a function */
#include<stdio.h>
 void addGraceMarks(int *m)
     { *m = *m+10;
     }
 void main( )
     {
       int marks;
       printf("Enter actual marks: It");
       scanf("%d",&marks);
       add GraceMarks(&marks);                          \* Passing address *\
       printf("\n Te graced marks is: \t%d", marks);
     }
```

* Program to convert upper case letter into lower and vice versa using passing pointer to a function *\

```
#include<stdio.h>
  void conversion(char*);                        \ function prototype *\
  main( )
     {
        char input;
        clrscr( );
        printf("Enter character of our choice:\n");
        scanf("%c",&input);
        conversion(&input);
        printf("\n The corresponding character is:\t%c",input);
        getch( );
     }
 void conversion(char*c)
     {
        if(*c>=97 && *c<=122)
              *c=*c-32;
        else if c *c>=65 && *c<=90)
              *c=*c+32;
     }
```

Output:

Enter character of our choice   : a

The corresponding character is   : A

9.4 Relationship between Arrays and Pointers:

There is a close association between pointer and array. Array name is pointer to itself. For example:

```
        int a[5],*P;
        P=&a[0];
```

After statement p=&a[0]; p point the array i.e. the p contains the address of a[0] (first address of array)

```
        P++;
```

After this statement P points to a[1] element.

```
#include<stdio.h>
#include<stdio.h>
  main( )
     {
        int a[5] = {5, 6, 7, 8, 9};
        int *P, i;
        clrscr( );
        p + &a[0];
     for(i=0, i<=4; i++)
```

```
        printf("%d\n",*(p+i));
        getch( );
      }
```
The output of above program is
```
      5
      6
      7
      8
      9
```
Because the statement printf("%d\n",*(p+i)); exzecute 5 times from i=0 to i=4. When i=0 then *(p+i) is *(p+0) means *p therefore this prints the value which store at location P.

Similarly *(p+1) means the value of second element because p+1 is pointer to the second element of array. Similarly, other elements are printed.

We know that array name is also pointer to so the above program can be written as

```
#include<stdio.h>
#include<conio.h>
 main( )
    {
      int a[5]={ 5, 6, 7, 8, 9};
      int i;
      clrscr( )
      for(i=0; i<=4; i++)
      printf("%d\n",*(a+i);
      getch( );
    }
```
output:
```
      5
      6
      7
      8
      9
```

Pointer and Multidimensional Array:

We know that the name of array is point to first element of array i.e. Oth element. In two dimensional array the array point to [0] [0] element. In three dimensional array the array name point to [0] [0] [0] element, etc.

In two dimensional array the second element is [0] [1].

If we want to evaluate the [1] [2] element through pointer then the expression for that is *(*(p+1)+2); where p is pointer to array (i.e. p contains the address of [0] [0] element) or p is name of array.

#include<stdio.h>

125

```
#include<conio.h>
  void main( )
    {
      int a[2] [3] = {{5, 6, 7},{8, 9, 10}};
      int i, j;
      clrsc( );
      for(c=0; i<=1; i++)
      {       for(j=o; j<2; j++)
              Printf("%d\t", *(*(a+i)+j));
      }
        printf("\n");
      }
    getch( );
      }
```
output:

5	6	7
8	9	10

9.5 Dynamic Memory Allocation (DMA):

The process of allocating and freeing memory at run time is k\a Dynamic Memory Allocation. This reserves the memory required by the program and returns this resource to the system once the use of reserved space utilized.

Through arrays can be used for data storage, they are of fixed size. The programmer must know the size of the array or data in advanced while writing the program. In some cases, it is not possible to know the size of the memory required well ahead and to keep a lot of memory reserved is not also good practice. In such situation, DMA will be useful.

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 to represent an array requires some type of initial memory assignment before the array elements are processed. This is k\a DMA. DMA refers allocating and freeing memory at run time.

There are 4 library functions malloc( ), calloc( ), free( ) and realloc( ) for memory management. Theses functions are defined within header file stdlib.h and alloc.h

1) 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_typr* malloc(size_of_block);

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

For example:

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

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

2) calloc( ):

The function calloc( ) provides access to the C memory heap which is available for dynamic allocation of variable_size block of memory. Unlike malloc( ), the function calloc( ) accepts two arguments: no_of_blocks and size_of_block. This parameter no_of_blocks specifies the number of items to allocate and size_of_block specifies the size of each item. The function calloc( ) allocates multiple blocks of storage, each of the same size and then sets all bytes to zero. One important difference between malloc( ) and calloc( ) is that calloc( ) initializes all bytes in the allocated block to zero. Thus, it is normally used for requesting memory space at runtime for storing derived data type such as arrays user defined. Its syntax is

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

For example:

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

or x=(int*)calloc(5,20);

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.

3) free( ):

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

Free(ptr);

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

4) realloc( ):

This function is used to modify the size of previously allocated space. Sometimes, the previously allocated memory is not sufficient, we nned 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 is done by the statement,

ptr=malloc(size);

then reallocationof space may be done by the statement

ptr=realloc(ptr,newsize);

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

```
\* Program to illustrate the use of realloc( ) *\
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
  void main( )
      {
        char*name;
        clrscr( );
        name=(char*)malloc(11);
        strcpy(name,"B. M. Singh");
        printf("\n Name=%s", name);
        name=(char*)realloc(ame, 18);
        strcpy=(name,"captain B. M. Singh");
        pritf("\nName=%s",name);
        getch( )
      }
```

output:

     Name   =   B. M. Singh

     Name   =   Captain B. M. Singh

```
/* Program to read marks obtained by dit.student & calculate sum & average using pointer */
#include<stdio.h>
#include<stdio.h>
#include<conio.h>
  main( )
      {
        int n, i;
        float *p, sum=0, avg;
        clrscr( );
        printf("\n How Many Students arethere?\t");
        scanf("%d",&n);
        printf("\n Enter marks of each students\n");
        p=(float*)malloc(n*sizeof(float));
        for(i=0; i<n; i++)
      {
        scanf("%f",(p+i));
        sum+=*(p+i);
```

```
        }
        avg=sum/n;
        printf("The average marks of");
        for(i=0; i<n; i++)
                printf(" % .2f \t " , *( p + i));
                printf(" % .2f \t is " , avg);


        free(P);
        getch( );
        }
```

output:

How many students are there? 5 ↵

Enter marks of each student

45.5   56   89   90.5   47

The average marks of   45.5, 56.00, 89.00, 90.50, 47.00   is   65.00

/* write a pg to read an array of n integers using dynamic memory allocation and display the largest and smallest element */

```
    int main( )
    {
      int n, i;
      int*num,max,min;
      clrscr( );
      printf("\n enter number of elements in our array = ");
      scanf("%d",&n);
      num = (int*)calloc(n,sizeof(int));
      printf("\nEnter%d integers=",n)
      for(i=0, i<n; i++)
      scanf("%d", num+i);
      max=*num;
      min=*num;
      for(i=0; i<n; i++)
      {
              if(max<*(num+i))
                 max=*(num+i);
              if(min>*(num+i))
                 min=*(num+i);
      }
      printf("\n the maximum number=%d",max);
      printf("\n the minimum number=%d",min);
      getch( );
      return( );
```

129

```
        }
```

output:

     Enter numbers of element in our array = 5

     Enter 5 integers = 12   67   89   34   32

     The maximum number = 89

     The minimum number = 12

Some Important Questions:

1.What are the relationship between arrays and pointer? Explain with example.

2.Write a program to accept 10 numbers and sort them with use of pointer.

3.Write a program to calculate sum of all number in a given matrix using pointer.

4.Defien an array and a pointer. How are they related?

5.What is a pointer? How is a pointer variable different from an ordinary variable?

6.Write a program to read 10 numbers and sort them in ascending order using pointer.

7.Write short notes on : Dynamic memory allocation

8.Write a function using pointers to add two matrices.

9.What is pointer? How is a pointer different from an ordinary   variable?

        10.What does p, &p and *p represents if 'p' is declared as integer pointer?

Chapter – 10

Structure and Union

Structure:

      Structure is a collection of data item. The data item can be different type, some can be int, some can be float, some can be char and so on. The data item of structure is called member of the structure.

      In other words we can say that heterogeneous data types can be grouped to form a structure. In some languages structure is known as record.

      The different between array and structure is the element of an array has the same type while the element of structure can be of different type. Another different is that each element of an array is referred to by its position while each element of structure has a unique name.

10.1 Defining a structure, arrays of structures, structures within structures:

      The general syntax for declaration of a structure is

```
storage_class   struct   name   {
data_type1      member1;
data_type2      member2;
- - - - -
- - - - -
data_typen      membern;
};
```

      where the storage_class is optional. The struct is a keyword. The name is written by the programmer. The rule for writing name is rule for identification. For example:

```
1) struct  student{
        char name[80];
        int roll_no;
        char branch[10];
        int semester ;
        };
2) static struct data {
        int day;
        char month_name[15];
        int year;
        };
```

Creating structure variable:

          131

In above section, we have studied about how we declare structure but have not created the structure variable. Note that the space in the memory for a structure is created only when the structure variable are defined.

Method of creating structure variable:

I. creating structure variable at the time of declaration:

Example:

struct student
{ char name[80];
  int roll_no;
  char branch[10];
  int semester;
} var1, var2, var3;

In this example three structure variables name var1, var2, var3 are created. We can create one or more than one variable. The space is created like:

var1

	name
	roll_no
	branch
	semester

var2

	name
	roll_no
	branch
	semester

var3

	name
	roll_no
	branch
	semester

II. struct student {   /* structure declaration */
    char name[80] ;
    int roll_no;
    char branch[10];
    int semester;
    };

/* structure variable creation */

struct student var1, var2, var3 ;

Accessing member of structure:

The accessing concept of member is:

structure_variable_name. member. The dot(.) operator is also known as as member operator or period. The dot operator has precedence and associativity is left to right. For example:

++var1. mark is equivalent to ++(var1.marks)

Implies that the dot operator acts first and thenunary operator.For example:

```
struct bio
{ char name[80];
   char address[80];
   long phno;
};
struct bio b1, b2;
for accessing
   b1.phno = 271280;
```

Initialization of structure:

A structure is initialized like other data type in C. The values to be initialized must appear in order sa in the definition of structure within braces and separated by commas. C doesnot allow the initialization of individual structure member within its definition. Its syntax is

struct structure_name structure_variable = {value1, value2, value3 _ _ _ _ valuen}

(1) struct particular{
            int, rn ;
            int age ;
            char sex; };

We can initialize the structure at the time of variable creation like:

struct_particular per = {10, 22, 'm'};

By this 10 is assigned to m of per; 22 is assigned to age of per and m is assigned to sex of per structure.

(2) struct bio
    { int age;
      int rn;
      char sex;
      int phno;
    };

133

If we write the following statement:

strwt bio b = {22, 10, 'm'}

In above example, C compiler assigns following value to each member:

age = 22	sex = m
rn = 10	phno = 0

So, C compiler will automatically initialized zero to those members who are not initialized.

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

```
# include <stdio.h>
    main( )
    {
      struct student
       {
       char name[20];
       int roll;
       float marks;
       char remark;
       };
    struct student s;
      clrsc( );
      printf("enter name: \t");
      gets(s.name);
      printf("In enter roll:\t");
      scanf("%d", &s.roll);
      printf("\n enter marks: \t");
      scanf("%f", &s.marks);
      printf("enter remarks p for pass or f for fail: \t")
      s.remark = getch( )
      printf("\n\n The student's information is \n");
      printf("Student Name ItItIt Roll It Marks It Remarks");
      printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
      printf("%s\t\t%d\t%.2f\t%c", s.name, s.roll, s.marks, s.remark);
      getch( );
    }
```

output:

enter name: Ram Singh

enter roll   : 45

entr marks : 89

Enter remark p for pass or f for fail : P

The student's information is

Student Name	Roll	Marks	Remarks
Ram Singh	45	89.00	P

Array of Structure:
Like array of int, float or char type, there may be array of structure. In this case, the array will have individual structure as its elements. For example:

```
struct employee
    { char name[20];
    int empid;
    float salary;
    } emp[10];
```

Where emp is an array of 10 employee structures. Each element of the array emp will contain the structure of the type employee. The another way to declare structure is

```
struct employee{
    char name[20];
    int empid;
    float salary;
    };
    struct employee emp[10];
```

/* Create a structure named student that has name, roll, marks and remarks as members. Assume appropriate types and size of member. Use this structure to read and display records of 5 student */

```
    main()
    {
      struct student
      {charname[30];
      int roll;
      float marks;
      char remark;
    };
struct student st[5];
    int i;
    clrscr( );
    for(i=0; i<5; i++)
    {
```

```
        printf("\n Enter Information of student No%d\n"; i+1);
        printf("Name: \t");
        scanf("%s", s[i].name);
        printf("\n Roll: \t");
        scanf("%d", &s[i].roll);
        printf("\n Marks:\t");
        scanf("%f", &s[i].marks);
        printf("remark(p/f): \t");
        s[i].remark = getch( );
    }
        printf("\n\n The Detail Information is \n");
        printf("Student Name: \t Roll \t Marks \t Remarks")
        printf("\n_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ \n")
    for(i=0; i<5; i++)
        printf("%s\t\t%d\t%.2f \t%c\n".s[i].name, s[i].roll, s[i].marks, s[i].remark);
        getch( );
    }
```

output:
Enter Information of student No1
    Name: Ram
    Roll: 20
    Marks: 89
    Remarks: P
Enter Information of Student No2
    Name: Shyam
    Roll: 21
    Marks: 46
    Remarks: P
Enter Information of Student No 3
    Name: Bikash
    Roll: 22
    Marks: 97
    Remarks: P
Enter Information of Student No 4
    Name: Jaya
    Roll: 23
    Marks: 87
    Remarks: P
Enter Information of Student No 5
    Name: Nisha
    Roll: 24

Marks: 27

Remarks: F

The Detail Information is:

Student Name	Roll	Marks	Remarks
Ram	20	89	P
Shyam	21	46	P
Bikash	12	97	P
Jaya	23	87	P
Nisha	24	27	F

Initializing array of structure:

Array of structure can be initialized in the same way as a single structure. For example:

```
Struct Book
{ char name[20];
  int pages;
  float price;
};
  Struct Book b[3] = {
"Programming In C", 200, 150.50,
"Let Us C", 455, 315,
"Programming with C", 558, 300.75}
```

Structure within structure (Nested Structure):

One structure can be nested within another structure in C. In other words, the individual members of a structure can be other structure as well. For example:

```
/* Create a structure named date that has day, month and year as its members. Include this structure as a member in another structure named employee which has name, id, salary, as other members. Use this structure to read and display employee's name, id, dob and salary */
#include<stdio.h>
 void main( )
    {
      struct date
      {
            int day;
            int month;
            int year;
      };
      struct employee
      {
```

137

```
        char name[20];
        int id;
        struct dat dob;
        float salary;
    } emp;
    printf("Name of Employee: \t");
    scanf("%s", emp.name);
    printf("\n ID of employee: \t");
    scanf("%d", & emp.id);
    printf("\n Day of Birthday: \t");
    scanf("%d", &emp.dob.day);
    printf("\n month of Birthday: \t");
    scanf("%d", &emp.dob.month);
    printf("\n Year of Birthday: \t");
    scanf("%d", &emp.dob.year);
    printf("salary of Employee: \t");
    scanf("%d", &emp.salary);
    printf("\n\n The Detail Information of Employee");
    printf("\n Name \t id \t day \t month \t year \t salary");
    printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n");
    printf("%s\t%d\t%d\t%d\t%d\t%.2f",emp.name,emp.id,emp.dob.day,    emp.dob.month,
        emp.dob.year, emp.dob.salary);
    }
```

output:

Name of Employee	:	Teena
Id of Employee	:	2001
Day of Birthday	:	10
Month of Birthday	:	10
Year of Birthday	:	1987
Salary of Employee	:	12000

The Detail Information of employee:

Name	ID	Day	Month	Year	Salary
Teena	2001	10	10	1987	12000.00

10.2 Processing a Structure:

(PU2008)

/* WAP to read records of 5 employee(Enter relevant fields: Name, address, salary, Id)
Display the records of the top three employees which have the highest salary */

```
void main( )
    {
      struct employee
    { char nam[20];
      char Address[20];
      float salary;
      int ID;
    };
      int, i, j ;
      float temp;
      struct emp[5];
      clrsc( ); printf("enter5employee Information: ");
      for(i=0; i<4; i++)
    { for(j=i+1; j<5; j++)
      {if(emp[i].salary<emp[j].salary)
              {temp = emp[i].salary; emp[i].salary = emp[j].salary; emp[j].salary = temp;}
      printf("Information of 3 employees having highest salary: In");
      printf("\nName\t\t\tAddress\t\tSalary\t\D\t\n");
      printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \n');
      for(i=0, i<3; i++)
    { printf("%s\t\t%s\t\t%.2f\t%d",    emp[i].Name,    emp[i].Address,    emp[i].Salary,
      emp[i].ID)
    }
      getch( )
    }
```

PU2007
/* Create a user defined array structure student record having members physics, chemistry
and mathematics. Feed the marks obtained by three students in each subjects and calculate
the total marks of each student */

```
void main( )
    { struct student
      {int physics;
        int chemistry;
        int mathematics;
        int total;
    };
    struct student std[3]; int i;
    printf("Enter the marks in physics, chemistry and mathematics of 3 student: ");
    for(i=0; i<3; i++)
    { printf("Marks of students %d", i+1);
      scanf("%d%d%d", std[i].physics, std[i].chemistry, std[i].mathematics);
```

139

```
std[i].total = std[i].physics + std[i].chemistry + std[i].mathematics;
}
printf("Information of students: ");
printf(\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \n");
printf("student\t\ttotal marks \t");
printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \n");
for9i=0; i<3; i++)
{
   printf("\nStudent%d\t\t%d\n", i+1, s+d[i].total);
}
```

10.3 Structures and pointers:

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

```
struct book
{ char name[20];
   int page;
float price;
};
```

Then we can define structure variable and pointer variable of structure type
book b ; /* b is structure variable*/
book *p; |* p is pointer variable of structure type *|

Where b is simple variable of structure type book where as p is pointer type variable which points or can store address of structure book type variable.

This declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer. To use structure's members through pointer p, memory must be allocated for a structure by using function malloc( )or by adding declaration and assignment as given below

p = &b;

Here, the base address of b can assign to p pointer.

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

ptr_variable ->member

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

i.e.

Now, the members name, pages, and price of book can be accessed as

b.name or p ->name or (*p).name

b.pages or p->pages or (*p).pages

b.price or p->price or (*p).price

```
#include<stdio.h>
#include<conio.h>
 void main( )
     {
       struct book;
       {char name[20];
       intpages;
       float price;
     };
     struct book b, *p;
     clrscr( );
     printf("Enter Book's Name: \t");
     gets(b.name);
     printf("\n Number of pages: \t");
     scanf("%d", &b.pages);
     printf("\n price of the book:\t");
     scanf("%+", &b.price),
        p = &b;
     printf("\n \n Book Information Using with arrow operator");
    printf("\n Book Name=%s\t Pages=%d\t price=%.2f, p->name, p-> pages, p->price);
     printf("\n\n Book Information using pointer with * operator");
     printf("\n   Book   Name=%s\t   Pages=%d\t   price=%.2f", (*p).name, (*p).pages,
          (*p).price);
     printf("\n\n Book Information using structure variable (i.e. dot operation)");
     printf("In Book Name=%s\t Pages=%d\t Price=%.2f", b.name, b.pages, price);
     getch( );
     }
```

Output:

Enter Book's Name   :   Let Us C

Nuber of Pages   :   540

Price of the Book   :   320.6

Book Information Using Pointer with arrow-operator

Book Name=Let Us C   Pages=540   Price=320.60

Book Information Using Pointer with * operator.

Book name=Let Us C   Pages=540   Price=320.6

Book Information Using Structure variable(i.e. dot operator)

Book Name=Let Us C   Pages=540   Price=320.6

141

10.4 Passing Structure to functions:

Like any other variable, a structure variable can also be passed to a function. We can either pass individual structure elements or the entire structure. The structure can be passed like following:

```
#inclde<stdio.h>
 struct A
      { int n1;
        float n2;
      };
void output (struct A a1);
void main( )
      {
      struct A a1;
        - - - - -
        - - - - -
        - - - - -
      output(a1);
        - - - - -
        - - - - -
      }
      void output struct (A a2)
      {
        - - - - -
        - - - - -
      printf("%d\n", a2.n1);
      printf("%f\n", a2.n1);
      }
```

/* WAP which reads two complex numbers and then perform multiplication of these two */

So|^n: If two complex no. are a1+ib1 and a2+ib2 then multiplication is

$$(a1+ib1)*(a2+ib2)$$
$$\Rightarrow a1a2 + i^2 b1b2 + a1b2i + a2b1i$$
$$\Rightarrow (a1a2 - b1b2) + (a1b2 + a2b1)i$$
$$(\text{because } i^2 = -1)$$

```
#include<stdio.h>
#include<conio.h>
 struct complex
      { float realpart;
        float imagpart; };
      struct complex mul ( struct complex, struct complex);
      void main( )
```

```
    {
    struct complex a, b, c ;
    clscr( );
    printf("Enter firstcomplex no.: \n");
    printf("Enter real part:\t");
    scanf("%f", &a.realpart);
    printf("Enter Image part: \t");
    scanf("%f", &a.imagpart);
    printf("Enter Second Complex no.:\n");
    printf("Enter real part:\t");
    scanf("%f", &b.realpart);
    printf("Enter imag part: \t");
    scanf("%f", &b.imagpart);
      c = mul(a,b);
    printf("First no. is: %f+i%f\n", a.realpart, a.imagpart);
    printf("Second no. is: %f+i%f\n", b.realpart, b.imgpart)
    printf("Multiplication is: %f+i%f\n", c.realpart, c.imagpart);
    getch( );}
    structcomplex mul (struct complex a, struct complex b)
    { structcomplex c;
       c.realpoint = (a.realpoint * b.realpart) – (a.imagpart * b.imagpart);
       c.imagpart = (a.imagpart * b.realpart) + (b.imagpart * a.realpart);
    return(C);
    }
```

output:

Enter first complex no.	:	
Enter real part	:	U2
Enter image part	:	4
Enter Second Complex no.	:	
Enter real part	:	U3
Enter image part	:	5
First no. is	:	2.000000 + i4.000000
Second no. is	:	3.000000 + i5.000000
Multiplication is	:	-14.000000 +i22.00000

10.5 Union and its important:

Union is similar to structure data type but union store value of different types in a single location. Union will contain many different type of values but only one is stored at a time. If a new assignment is made, the previous value is automatically erased.

The declaration of union is same as the structure. Only difference is in place of struct keyword the union keyword is used. The accessing of union member is also same like structure member. For example:

```
Union data {   int a;
                 float b;
              };
```

The union variable are declared like structure variable. For example:

```
Union data d1, d2;
```

If we write the statement

```
d1.a = 10;
```

Then that means the field a is valid but we want to print the value of b after above assignment, the value is wrong because a is valid only. After above assignment if we write d1.b = 20.5; now b is valid not a i.e. only that field value is valid in which we have assigned value currently.

Space Created for Union

We know that union store one value at a time so the question is now much space is created for union. C compiler first calculates the size of all members and then reserves the space which is highest among them. For example:

```
Union A{      int i ;
                 float f ;
          char c; };
          union A a;
```

The space is created for a is 4 byte. Because I needs 2 bytes, f needs 4 bytes and c needs 1 byte, 4 bytes is highest in these.

```
/* Create a union named student that has roll and marks as member 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 = 80;
    printf("\nMarks=%f", st.marks);
    }
```

output:

    Roll=455
    Marks=80.000000

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

```
void main( )
    {
        union student
        {introll;
        float marks;
        };
    union student st;
        st.roll = 455;
        st.marks = 80;
    printf("\nRoll=%d", st.roll);
    printf("\nMarks=%f", st.marks);
    }
    Output:   Roll = 0
              Marks = 80.000000
```

Where, roll is zero as memory is replaced by another variable st.marks.

Difference between structure & unoion:

	Structure		Union
(1)	Each member within a structure is assigned its own unique storage. It takes more memory than union.	(1)	All members within union share the same storage area of computer memory. It takes less memory than structure.
(2)	The a\m of memory required to store a structure is the sum of the size of all members.	(2)	The a\m of memory required to store an union is same as member that occupies largest memory.
(3)	All the structure members can be accessed at any point of time.	(3)	Only one member of union can be accessed at any given time.

(4)	Structure is declared as
	struct student
	{ char name[20];
	  float marks;
	} st;
		(4)	Union is declared as
			union student
			{ char name[20];
			  int roll;
			  float mark;
			} st;

Some Important Questions:

1. What are structures? How and when the they declared in C-program?

2. Write a program having a structure of student type. Make use of array of structure to input information of 20 students.

3. How structure members are accessed using pointer. Distinguish between Structure and union.

4. Write a program to read the name, roll no and name of five students using array of structure object. Display the name and roll no of those students mark is greater than 50.

5. Write a program to read several different names , roll, address, percentage and display name who has score the 3rd highest.

6. Create a user defined array structure student record having member's physics, chemistry and mathematics. Feed the marks obtained by three students in each subjects and calculate the total of each student.

Chapter – 11

Data Files

The input output function:

Like printf( ), scanf( ), getchar( ), putchar( ), gets( ), Puts( ) are known as console oriented I/O functions which always use keyboard for input device. While using these library functions, the entire data is lost when either the program is terminated or the computer is turned off. At the same time, it is cumbestsome and time consuming to handle large volume of data through keyboard. It takes a lot of time to enter the entire data. If the user makes a mistake while entering the data, he/she has to start from the beginning again. If the same data is to be entered again at some later stage, again we have to enter the same data. These problems invite the concept of data file 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:

1. High level (standard or stream oriented) files.
2. Low level (system oriented) files.

In high level data files, the available library functions do their own buffer management where as the programmer should do it explicitly in case of lower level files.

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. For each, binary and text files, there are a number of formatted and unformatted library functions in C.
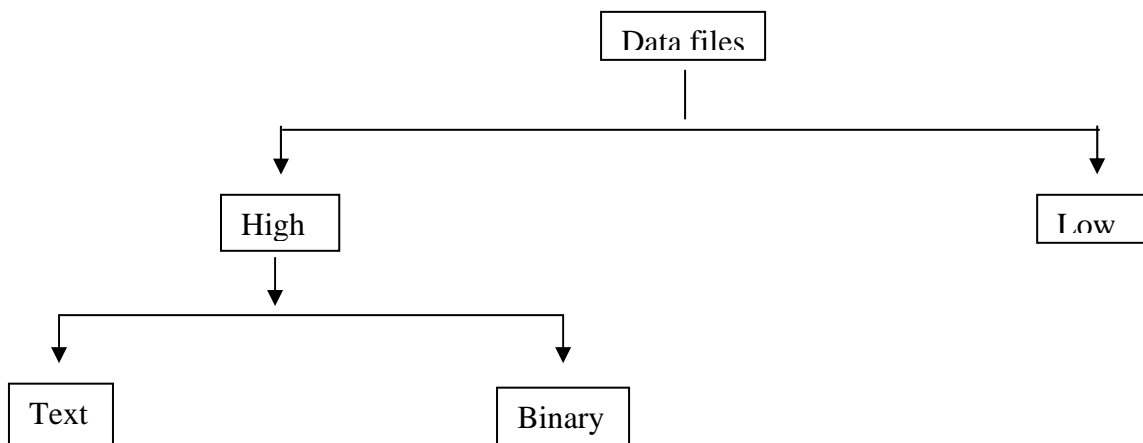

fig: classification of files

11.1 Opening and closing a data file:

Before a program can write to a file or read from a file, the program must open it. Opening a file established a link between the program and the operating system. This provides the operating system, the name of the file and the mode in which the file is to be opened. While working with high level data file, we need buffer area where information is stored temporarily in the course of transferring data between computer memory and data file.

The process of establishing a connection between the program and file is called opening the file.

A structure named FILE is defined in the file stdio.h that contains all information about the file like name, status, buffer size, current position, and of file status, etc. All these details are hidden from the programmer and the operating system takes care of all these things.

typedef struct

```
{
  - - - - - -;
  - - - - -;
  - - - - -;
} FILE;
```

A file pointer is a pointer to a structure of type FILE. Whenever a file is opened, a structure of type FILE is associated with it, and a file pointer that points to this structure identifies this file. The function fopen( ) is used to open a file.

The buffer area is established by

FILE *ptr_variable;

And file is opened by using following syntax

Ptr_variable = fopen(file_name, file_mode);

where fopen( ) function takes two strings as arguments, the first one is the name of the file to be opened and the second one is file_mode that decides which operations (read, write, append, etc) are to be performed on the file. On success, fopen( ) returns a pointer of type FILE and on error it returns NULL. For example:

FILE   *fp1, *fp2 ;

fp1 = fopen ("myfile.txt", "w");

fp2 = fopen ("yourfile.dat", "r");

The file opening mode specifies the way in which a file should be opened (i.e. read, write, append, etc). In other word, it specifies the purpose of opening a file. They are:

1. "w" (write):

If the file dosen't exist then this mode creates a new file for writing, and if the file already exists, then the previous data is erased and the new data entered is written to the file.

2. "a" (append):

      If the file doesn't exist then this mode creates a new file, and if the file already exists then the new data entered is appended at the new data entered is appended at the end of existing data. In this mode, the data existing in the file is not erased as in "w" mode.

3. "r" (read):

      This mode is used for opening an existing file for reading purpose only. The file to be opened must exist and the previous data of file is not erased.

4. "w+" (write + read):

      This mode is same as "w" mode but in this mode we can also read and modify the data. If the file doesn't exist then a new file is created and if the file exists then previous data is erased.

5. "r+" (read + write):

      This mode is same as "r" mode but in this mode we can also write and modify existing data. The file to be opened must exist and the previous data of file is not erased. Since we can add new data and modify existing data so this mode is also called update mode.

6. "at" (append + read):

      This mode is same as the "a" mode but in this mode we can also read the data stored in the file. If the file doesn't exist, a new file is created and if the file already exists then new data is appended at the end of existing data in this mode.

      To open a file in binary mode we can append 'b' to the mode and to open the file in text mode 't' can be appended to the mode. But since text mode is the default mode, 't' is generally omitted while opening files in text mode. For example:

"wb"	Binary file opened in write mode
"ab+" or ("a+b")	Binary file opened in append mode
"rt+" or ("r+t")	Text file opened in update mode
"w" or ("wt")	Text file opened in write mode

      The file that was opened using fopen( ) function must be closed when no more operations are to be performed on it. After closing the file, connection between file and program is broken. Its syntax is

      fclose(ptr_variable);

      On closing the file, all the buffers associated with it are flushed and can be available for other files. For example:

      fclose(fp1);

      fclose(fp2);

Library functions for reading/writing from/to a file:

1. <u>Unformatted I/O functions:</u>

a) <u>Character I/O functions:</u>

     <u>fget( ):</u> It is used to read a character from a file. Its syntax is

       char_variable = fget(file_ptr_variable);

     <u>fputc( ):</u> It is used to write a character to a file. Its syntax is

       fputc(char_variable, file_ptr, vriable);

b) <u>String I/O functions:</u>

     <u>fgets( ):</u> It is used to read string from file. Its syntax is

       fgets(string, int_value, file_ptr_variable);

     <u>fputs( ):</u> It is used to write a string to a file. Its syntax is

       fputs(string, file_ptr_variable);

2. <u>Formatted I/O functions:</u>

     <u>fprintf( ):</u> This function is used to write some integer, float, char or string to a file. Its syntax is

       fprintf(file_ptr_variable,"control string", list variables);

     <u>fscanf( ):</u> This function is used to read some integer, float char or string froma file. Its syntax is

       fscanf(file_ptr_variable,"control string", & list_variables);

11.2 Creating data files:

Some sample programs:

/* create a file named "test.txt" and write some text "Welcome to Eastern College of Engineering" to the file.*/

```
#include <stdio.h>
    void main( )
    {
            FILE * fp;
            fp = fopen("C:\\test.txt", "w");
            if(fp = = NULL)
    {
    else
    {
            printf("File has been successfully created");
    }
            fputs("Welcome to Eastern College of Engineering; fp);
    fclose(fp);
```

output: goto C:\ drive and see the text file test.txt in notepad where the content is Welcome to Eastern College of Engineering.

Er. Arun Kumar Yadav,
Lecturer, Eastern College of Engineering, Biratnagar

|* Program to open the file "text.txt" created above, read its content & display to the screen *|

```
#include <stdio.h>
     Void main( )
       {
              FILE * fp ;
              char s[100]
              fp = fopen("c:\\test.txt", "r");
              if (fp = = NULL)
              {
              printf("In file can not be opened");
              exit( );
              }
              fgets(s, fp);
              printf("in the text from file is: \t%s", s);
              fclose(fp);
              getch( );
       }
```

OUTPUT: The text from file is Welcome to Eastern College of Engineering.

|* Program that opens a file and copies all its content to another file. Take source & destination file from user *|

```
#include <stdio.h>
     Void main( )
     {
       FILE *fsource, *fdest;
       char, ch, source[20], dest[20] ;
       printf("In Enter source file name: \t");
       gets(source);
       printf("Enter destination file name: \t");
       gets(dest)
       fsource = fopen(source, "r");
       if(fsource = = NULL)
     {
       printf("\n source file can not be opened"),
       exit( );
     }
       fdest = fopen(dest, "w");
       if (fdest = = NULL)
     {
       printf("\n Destination file can't be created"),
```

151

```
        exit( );
    }
      while(( ch = fgetc(source)! = EOF)
    {
      fputc(ch, fdest);
    }
      printf("Sucessfully copied!");
      fclose(fsource);
      fclose(fdest);
      getch( ):
    }
```

|* Program to understand fprintf( ) *|
```
#include <stdio.h>
      Struct student
      { char name[20];
        float marks;
      } std;
      main( )
      { FILE *fp;
        int i, n;
        fp = fopen("student.dat", "w");
        printf("enter number of records:");
        scanf ("%d", &n);
        for(i=1, i<=n; itt)
      { printf("enter name and marks :");
        scanf("%s%f", std.name, & std.marks);
        fprintf(fp, "%s%f", std.name, std.marks);
      }
        fclose(fp);
      }
```

|* Program to understand fscanf( ) *|
```
#include <stdio.h>
      { char name[20];
        float marks;
      } std;
      main( )
      { FILE *fp;
        fp = fope("student.dat". "r");
        printf("Name \t Marks \n ");
```

```
    while (fscan(fp, "%s%f", std.name, & std.marks)! = EOF)
    printf("%s%f in", std. name, std.marks);
    fclose(fp);
}
```

End of File (EOF):

The file reading function need to know the end of file so that they can stop reading. When the end of file is reached, the opening system sends an end-of-file signal to the program. When the program receives this signal, the file reading function returns EOF, which is a constant defined in the file stdio.h and its value is -1.

Predefined File Pointer:

Three predefined constant file pointer are opened automatically when the program is executed.

File Pointer	Device
Stdin	Standard input device (keyboard)
Stdout	Standard OUTPUT devices (screen)
Stderr	Standard error OUTPUT device (screen)

Some other unformatted function:

Intger I|O   -   getw( ), putw( )[for binary mode]

Record I|O   -   fread( ), fwrite

Putw( ): used to write integer value th file. It returns the integer written to file on success and EOF on error

getw( ): used to read integer value from a file. It returns the next integer from the i|p file on success and EOF on error. Their syntaxes are:

        int putw(int value, FILE *fpir);

        int getw(FILE fptr);

fwrite( ): used for writing an etire block to a given file.

fread( ) is used to read an entire block from a given file. Their syntax are:

        fwrite(&ptr, size_of_array_or_structure, number_of_array_or_structure, fptr);

        fread(&ptr, size_of _array_or_structure, number_of_structure_or_array, fptr);

Random Acess to File:

We can acess the data stored in the file in two ways, sequently or random. So, far we have used only sequently acess in our programs. For example, if we want to acess the forty-fourth record then firsat forty three records should be read sequentially to reach the forty-forth record. In random acess, data can be accessed and processed randomly i.e. in this case the forty-forth record can be accessed directly. There is no need to read each record

sequentially, if we want to acess a particularly record. Random acess takes takes less time than the sequential acess.

  C supports these functions for random acess file processing:

   - fseek( )

   - ftell( )

   - rewind( )

fseek( ):

This function is used for setting the file position pointer at the specified byte. The syntax is

  int fseek(FILE *fp, long displacement, int origin)

where fp is file pointer, displacement is long integer which can be positive or negative and it denotes the number of bytes which are skipped backward (if negative) or forward (if positive) from the position specified in the third argument.

  The third argument named origin is the position relative to which the displacement takes place. It can take one of these three values:

Constant	Value	Poition
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of File

rewind( ):

 This function is used to move the file position positive to the beginning of the file syntax

  rewind(File *fp)

using rewind(fp) is equivalent to fseek(fp, OL, O)

ftell( ):

This function returns the current position of the file position pointer. The value is counted from the beginning of the file. The syntax is

  long ftell(FILE *fp);

Some Programs:

/* Program to write some text "welcome to my college" to a data file in binary mode. Read its content and display it */

```
#include <stdio.h>
    Void main( )
    { FILE *fptr;
      char c;
      fptr = fopen("test.txt", "w+b")
      if(fptr = = NULL)
    { printf("\n File can not be created");
      fputs("welcome to my college", fptr),
      rewind(fptr);
```

```
    printf("The content from file: \n");
    while( cc = fget c ffptr))! =EOF)
    {
            printf("%c", C);
    }
            fclose(fptr);
    }
```

output: The Content from file:
    Welcome to my college.

® |* Create a structure named employee having numbers: empName, age and salary. Use this structure to read the name, age and salary of employee and write entered information to a file employee.dat in D:\ drive *|

```
#include <studio.h>
    void main( )
    {
      struct employee
    { char empName[20];
      int age;
      float salary;
    };
      struct employee emp;
      FILE *fptr ;
      ftpr = fopen("d:\\employee.dat", "wb");
      if(fptr = = NULL)
      { printf("File can not be entered");
        exit( );
      }
      printf("Employee Name: It");
      scanf("%s", emp. empName);
      printf("employee age: It);
      scanf("%d", &emp.age);
      printf("salary of the employee: It");
      scarf("%f", & emp.salary);
      printf("In writing this information to a file _ _ _ In");
      fwrite(&emp, sizeof(emp), 1, fptr);
      getch( );
    }
```

|*Create a structure named student that has name, roll and marks as members. Assume appropriate types and size of member. Use this structure to read and display records of 3 students. Write an array of structure to a life & then read its content to display to the screen *|

```
Void main( )
    { struct student
      {Char name[30]; int roll; float marks;};
      struct student s[3], st[3];
      int i,
      float tempMarks;
      FILE *fptr ;
      fptr = fopen("d:\\student.txt". "w+b")
      if(fptr = = NULL)
      { printf("File can't be created.");
      exit( );
      }
      for(i=0, i<3; itt)
      { printf("\n Enter Information of student No %d \n", i+1);
      printf("Name: \t"); scanf("%s", s[i].name);
      printf("\n Roll:\t"); scanf("%d", & s[i].roll);
      printf("\n Marks: \t"); scanf("%f", & tempMarks);
        s[i].marks = tempMarks;
    }
      printf("\n working Information to file _ _ _ _ _ \n");
      fwrite(&s, size of (s), 3, fptr);
      rewind(fptr);
      printf("\n reading same content from file _ _ _ _ _ \n");
      fread( &st, size of (st), 3, fptr);
      printf("\n student Name \t Roll \t Marks");
      printf("\n _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ \n")
      for(i=0, i<3; i++)
      printf("%s\t\t%d\t%.2f \n", st[i].name, st[i].roll, st[i].marks);
      fclose(fptr);
      getch( );
    }
```

In Program ® read employee information again and again until user wants to add more employees. Finally, write a program to search information of a particular employee from the file.

```
#include <stdio.h>
#include <string.h>
    void main( )
```

Computer Programming in C

```
{ struct employee
  {char name[20]; int age; float salary;
  };
  structemployee emp;
  FILE *fptr;
  char yes_no, name[15];
  int dataFound = 0;
  clrscr( );
  fptr = fopen("C:\\employee.txt", "w+b"),
  if(fptr = = NULL)
  { printf("File can not be created");
  exit( );
  }
  Do { printf("employee Name: \t");
  scanf("%s", & emp.name);
  printf("employee age: \t");
  scanf(%d", &emp.age);
  printf("salary of the employee: \t");
  scanf("%f", &emp.salary);
  fwrite(&emp, size of (emp), 1,fptr0,
  printf("Do you want to add another employee? Press Y or Y: it);
  fflush(stdin);
  yes_no = getchar( );
  3 while(yes_no = = 'y' || yes_no = = 'y');
  printf("Enter the name of employee which is to be searched")
  fflush(stdin);
  gets(name);
  rewind(fptr);
  while(fread (&emp, size of (emp), 1 fptr) = = 1)
{ if(strcmp(emp.name, name) = = 0)
  {dataFound = 1;
  printf("Name \t Age \t \t Salarly \n")
  printf("\n _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ \n")
  print("%s\t%d\t%.2f",emp.name, emp.age, emp.salary);
}}
If(dataFound = = 0)
  print("\n Matching data not found.");
  getch( );
}
```

Difference between text and binary mode:

	Text mode		Binary mode
1.	Each line is terminated by '|h'.	1.	Data is stored on the disk in the same way as it is represented in computer memory.
2.	Text files are in human readable form and they can be created and read using any text editor.	2.	Binary files are not in human readable form and can be created and read by specific programs written for them.
3.	The integer 1679 will occupy 4 bytes because in a text file it is stored in a sequence of 4 characters i.e. '1', '6', '7', '9'.	3.	The integer will take only 2 bytes in binary file because in binary filr it is stored a s integer.
4.	It take more time for i|p, output operation compared to binary file.	4.	It takes less time for i|p output operation due to no need toconvert.

Some Important Questions :

1. Write a program to enter name, roll and mark of 10 students and store them in the file. Read and display the same from the file. (PU2007)

2.Write a program to create a data files containing record roll number and students name and total marks. (PU2006)

3.What is a files ad explain its importance in C programming.

4.What is a file? Write a program in C that allows a user enter name, age, sex, address, number of songs recorded duet singers using a file. The program should also have the provision for editing the details of any singer.

5.Describe any two file handling input/output function.

6.Write a program that read a line of text and store file then print the content of file.

7.Explain in brief the various file opening modes.

8.Explain in brief the steps involved in opening a file.   [

9.Write a program to open a file in write mode, take input form the keyboard, and write it to the file.

10.What is the significance of EOF?

Chapter – 12

Graphics

The C graphics function fall into two categories those that work in text mode and those that work in graphics mode.

The text mode functions are concerned with placing text in certain area of screen. They work with any graphics monitor and adapter.

The graphics mode function require a graphics monitor and adapter card such as CGA, EGA or VGA. Graphics mode function allow us to draw dots, lines and shapes (like circle, rectangle, etc.), add color to lines and area and perform many other graphics related activities.

In order to perform graphics related activities, we include graphics.h.

Some text mode graphics function:

clrscr( )   : This function erases the text window. Its syntax is clrscr( );

window( )   : This fuction takes four integer arguments that determine the left, top, right and bottom coordinates of the window. The general syntax:

Window (left, top, right, bottom)

gotoxy( )   : The gotoxy( ) library function points the cursor position within a text window; its syntax is gotoxy(x,y);

text color( ) function: It is used to give color of any text. The syntax of text color( ) is text color(color constant)

textbackground ( ) function: This function is used to set background color of each character. The syntax is

textbackground (color constant);

Some color constants:

Color No.	Color name
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown, etc.

12.2 Graphics Mode Graphic function:

Initialization:

In order to initialize graphics, we initgraph( ) function. This makes computer screen in specified graphics mode. This makes computer screen in specified graphics mode. This function initializes the graphics system by loading a graphics driver from disk then putting the system into graphics mode. Its general syntax is

initgraph (& graphics_driver, & graphics_mode, "path to driver");

where graphics_driver is a variable of type int initialized to some constant that defined in graphics.h. This variable specifies the graphics driver are applicable only in graphics mode and they communicate directly with monitor.

C offers certain graphics driver and these are the files with .BGI extension. They are stored in subdirectory BGI of TC directory (particularly C:\\T\\BGI). Depending on what adapter is used, one of these driver gets selected. The some constants defined in graphics .h file for this argument are DETECT(=0), CGA(=1), EGA(=3), VGA(=9),etc.

Closing Graphics Mode:

Once a program has finished its job using the graphics facilities, then it should restore the system to the mode that was previously in use (i.e. the graphics mode should be closed). If graphics mode is not closed explicitly by the programmer, undesirable effects may be felt. The closegraph( ) fuction is used to restore the screen to the mode it was in before we called initgraph( ) and deallocates all memory allocated by the graphics system. The syntax of closegraph( ) sunction is

closegraph( ) ;

Graphics Mode:

In text mode we are restricted to display text or graphics character but in graphics mode we can display points, lines and compleax shapes. In text mode, we can address only 2000 location (80*25) but in graphicsmode we can address individual pixel or dots on the screen. This gives we much finer resolution.

For example in a 640x480 VGA mode we can address 307,200 pixels. If we want to use graphics mode functions then we have to add graphics .h header file but for text mode function no need of graphics .h header file. If we want to use the graphics mode functions then our first job is to set our computer mode to graphics mode.

Function initgraph( ) sets our mode for graphics work. The syntax of initgraph( ) function is

initgraph(& driver, & mode, path of bgi files), where driver and mode both are integer type and path for bgi files means where our .BGI files are in disk.

Computer Programming in C

If we want to set the mode of computer is graphics mode by initgraph ( ) function then we have to write the following set of statements:

intdriver, mode;

driver=DETECT;

initgraph(& driver, & mode, "C:\\tc\\bgi")

we are assuming that our .bgi are in sub directory C:\tc\bgi therefore the path of bgi files is written in initgraph ( ) function like:

"C:\\tc\\bgi"

The statement driver=DETECT will check our hardware and select appropriate values for argument to function initgraph( )

Some Graphics Mode Graphic 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 a point 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 point with 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 x and y.

161

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

<u>rectangle( ):</u> The function rectangle (x1, y1, x2, y2) draw rectangle where point (x1, y1) is left top corner point of rectangle and point (x2, y2) is right bottom corner.
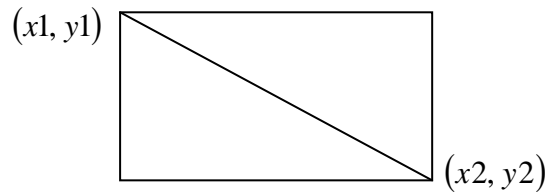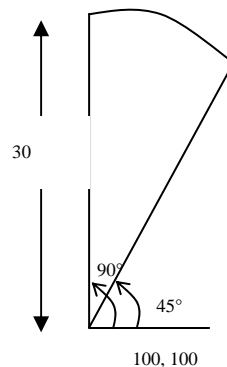

<u>circle( ):</u> The function circle (x, y, r) draws a circle of radius r. The coordinates of the centre of the circle is (x, y).
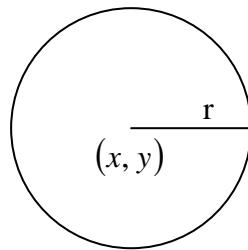

<u>arc( ):</u> Function arc (x, y, a1, a2, r) draws an arc on the screen starting from angle a1 to a2. The radius of the circle of which the arc forms a part is r and x, y are its centre coordinates. For example:

arc (100, 45, 90, 30) ;

Above statement draws an arc like

<u>ellipse( ):</u> 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.

162
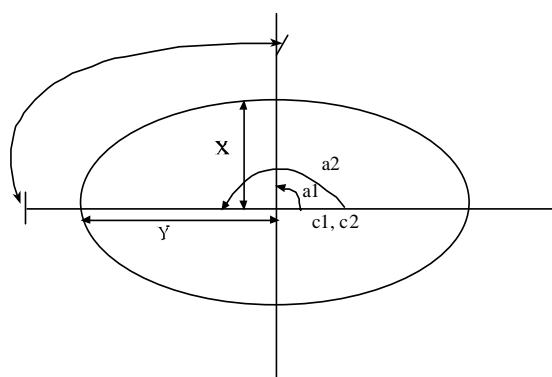
drawpoly( ): Function drawpoly (int n, int p[ ]); draws n vertices of polygon. P is an array of integer numbers which gives x and y co-ordinates of the points to be joined. To draw a closed polugon with n vertices, we must pass n+1 co-ordinates.

        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 if user defined style (USERBIT_LINE) is used. We can specify it to zero. The thickness parameter can have value NORM_WIDTH or THICK_WIDTH or integer value 1 or 3 respectively.

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

fillpoly (int n, int p[ ]);

To draw a closed polygon with vertices, we must pass n+1 co-ordinates to fillpoly( ).

12.3 Simple program using built in graphical function:

2005 PU

|* 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);
```

           163

```
        rectangle (0, 0, 100, 200);
        ellipse (150, 250, 0, 360, 80, 60);
        getch ( );
        closegraph ( );
    }
```

2007 PU
|* 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 ( );
    }
```

Some Important Questions:

1.Write a program to draw: (PU2003)

       (i) Line
       (ii) Circle
       (iii) Rectangle
       (iv) Arc.

2.Write a program using graphics to draw a rectangle and a circle. (PU2003 BACK)
3.Write a program in C to draw a circle, a rectangle and an ellipse. (PU2005)
 4.Write a program in C that can display a circle and a rectangle. Choose centre and radius of the circle as well as coordinates of the rectangle on your own. (PU2006)
5.Write a program to draw 3 concentric circles having radius 25, 50 and 75 units.   (PU2007)

<u>References:</u>

1.The C Programming Language,Kernighan and Ritchie,PHI.

2.Programming ANSI C ,E. Balagurusamy,Mc.GRAW HILL.

3.Computer Fundamentals,P.K.Sinha.

4.Let us C,Yashwant Kanitkar,BPB.

5.C in Depth ,Srivastawa and Srivastawa,BPB.

6.Programming with C ,Byron Gottfried, Tata Mc.GRAW HILL.

7.Programming in C ,S.K. Jha,Katson.

8.Thinking in C++ ,Sunil Pandey,Katson.