C PROGRAMMING NOTE

for BCT/BEX/BEL/BIT/BCA

Chapter-1
Introduction to C Programming

1.1 Historical Development of C

C programming language was developed in 1972 by Dennis Ritchie at Bell Laboratories of AT & T, located in the U.S.A. It was developed to overcome the problems of previous languages such as B, BCPL etc. Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

1.2 Importance of C

1. A rich set of built-in functions and operators can be used to write any complex programs.
2. Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators.
3. Concise and powerful 32 keywords.
4. Programs written in C are highly portable. That means the C programs written in one computer can be run on another computer with little or no modification.
5. C being structured programming, helps the programmer to think of problems in terms of functions modules and blocks.

1.3 Basic Structure of C programming

The structure of c program can be described as.

1. **Documentation Section**
2. **Link Section**

3. **Definition Section**
4. **Global declaration section**
5. **main() function section**

   Declaration Part

   Executable Part
6. **Sub program section**

   Function 1

   Function 2

   …….

   …….

   Function n

1.3.1 Documentation Section

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer like to use later.

1.3.2 Link Section

This section provides instructions to the compiler to link function from the system library. For example: **#include <stdio.h>**

1.3.3 Definition Section

This section defines all the symbolic constants. For example: **#define SIZE 20**

1.3.4 Global declaration section

The variables that needs to be known or accessible to all the part or functions of the program are declared global. In this section all the global variables are declared. Also, all the user-defined functions are declared.

1.3.5 main() function section

This is the main section of the program, that the program code actually starts from this function to execute(entry point of the program). This section consists of two parts declaration part and executable part.

- Declaration part

  In this section, all the variables that are used in executable part in the program are declared.
- Executable part

  This section, consists all the statements that actually do something in the program.

1.3.6 Subprogram section

This section, contains all the user-defined functions that are used in the main program.

1.4 Executing a C Program

Executing a C program involves the series of steps.

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

1.4.1 Program

This is the sequence of well organized statements to solve a particular program.

1.4.2 Source Code

This is the program written in human readable format.

1.4.3 Machine Code

The code that is in machine readable format/ binary format is called machine/ object code. The compiler converts the source code into machine code.

1.4.4 High level language

Program that is in human readable format.

1.4.5 Low level language

Program that can only understand by machine/ computer.

1.4.6 Compiler

A program or software that converts high level language (source code) into machine level language (machine code) is called compiler. And the process of converting high level language to machine level language is called **compilation process**.

1.4.7 Linker

This is a program or software which helps to link object modules of program into a single object file. It also links the particular module to a system library. And the process is called **linking.**

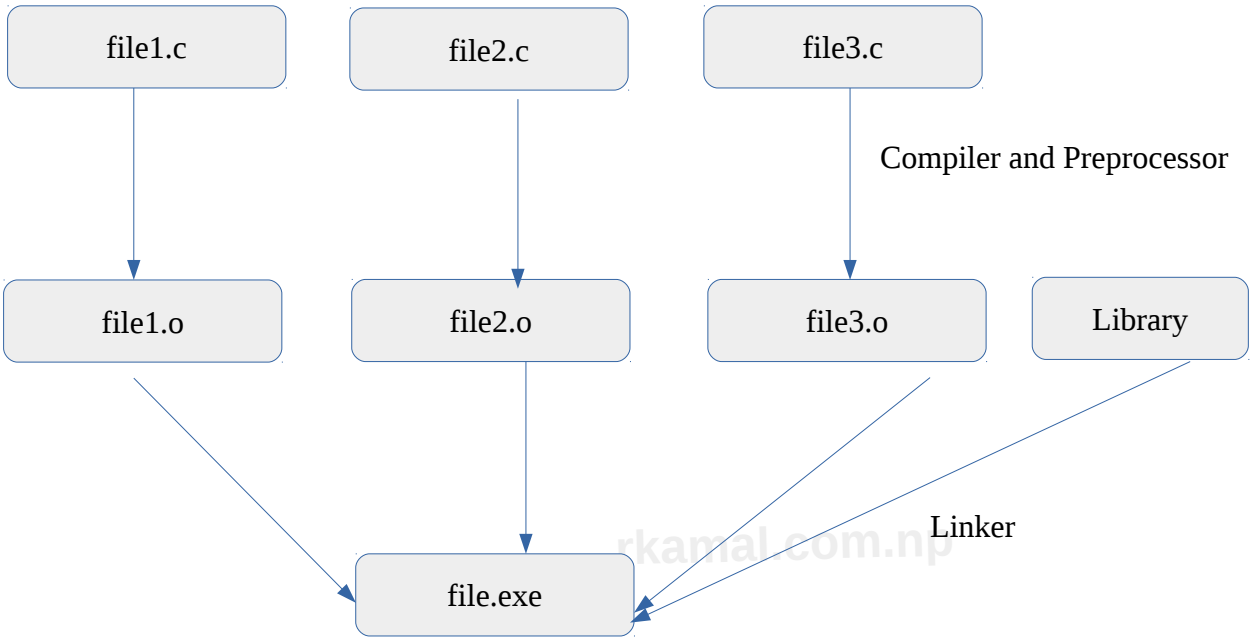


Fig. 1.4 Compilation of multiple files

1.5 Interpreter

-program that converts each high level program statement into a machine code just before the program is to be executed.
-Translation and execution occurs immediately one statement at a time.

Difference between Compiler and interpreter :

Compiler	Interpreter
1) Converts all the statements in source to object code and finally in executable code resulting an exe file.	1) It converts each statements before executing it . Does not produce exe file.
2) It requires some time before producing an executable file.	2) Can execute program immediately.
3) Once compiled then no need to be recompile for next run. So, the executables are much faster and efficient.	3) For next running it is required to repeat the process from the beginning.
4) Immediate editing and executing of the program is costly because it takes long time for compilation if the program is long.	4) Possible to execute the edited program immediately. So, used for software developing phase and learning for students.
5) eg. C, C++, FORTRAN etc.	5) eg. BASIC, Lisp, Javascript etc.

Firmware :
-Software embedded in hardware, that acts as interface between software and harware.
eg. The BIOS in computer, Device Drivers etc.

Chapter-2
Problem Solving Using Computer

-To solve clients requirements with computer based system, a good quality software needs to be developed. So we need to follow some process to accomplish our goal.

2.1) **Problem analysis :**

-also called problem definition.
-we have to give clear , concise problem statement.
-It should clearly specify the following.

i. Objectives
   -The problem should be stated clearly.
   -Simple program can be stated easily but complex program may need complex analysis.
ii. Output requirements
   -We should know what should come out from the system being developed.
   -Better to design the output requirements, sitting with the end users.
iii. Input requirements
   -To get the required output it is required to define the input data and the source of input data. eg. student data (from the college administration), survey data of road (from department of road).
iv. Processing requirements
   -Required to clearly define the processing requirements to convert the given input data to the required output. In processing requirement there may be hardware platform, software platform, manpower etc.
v. Evaluating feasibility
   -It is the phase where we decide whether the proposed software development task is technically and economically feasible.

2.2) **Algorithm development and flowcharting**

2.2.1 Algorithm development :

-Set of ordered steps or procedure necessary to solve the problem.
-It must be unambiguous and have a clear stopping point.
-verbal form of program.
-Each step tells what task is to be performed.

eg. An algorithm to find the area of the rectangle.

Step 1 : Start.
Step 2 : Accept the length of the rectangle.
Step 2 : Accept the breadth of the rectangle.
Step 2 : Multiply length and breadth of the rectangle in order to find the area of the rectangle.
Step 2 : Display the area.
Step 2 : Stop.

Guideline for algorithm development :

-Use plain language.
-Don't use any programming language specific syntax because same algorithm can be implemented using any programming language.
-Every job to be done should be described clearly without any assumptions.
-It must have single entry and exit point.

An excellent algorithm should have the following properties

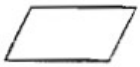
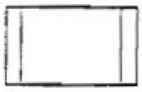
-finiteness : should have finite number of steps to solve the problem
-definite : Action of each step should be defined clearly without any ambiguity.
-Inputs : Inputs should be defined precisely
-Outputs : Each algorithm should result in one or more outputs.
-Effectiveness : It should be more effective among the different ways of solving the problem.

2.2.1 **Flowchart**

-Diagrammatic or pictorial representation of the algorithm.
-Generally drawn in the design stage of computer solution.
-Quite helpful in understanding the logic of complicated and lengthy problems.
-Helpful in explaining the programs to the others.

Symbols :

Table 2.1 Brief description of flowcharting symbols

Symbol	Purpose	Description
	Flow line	Used to connect symbols and indicates the flow of logic
	Terminal (start / stop)	Used to indicate the start and end of a flowchart. One flow line exits from start and one enters to stop.
	Processing	Used whenever data is being manipulated, most often with arithmetic operations. A single flow line enters and a single flow line exits.
	Input/Output	Used whenever information is entered into the flowchart or displayed from the flowchart. A single flow line enters and a single flow line exits.
	Decision	Used to represent operations in which there are two possible alternatives i.e. decision making and branching. One flow line enters and two or three flow lines (labeled true and false) can exit.
	Predefined process/ Function	Used to represent a function call, or variables are sent to another function to return data or an answer. A single flow line enters and a single flow line exits.
	On-page Connector	Used to connect remote flowchart portions on the same page. One flow line enters or exits.
	Off-page connector	Used to connect remote flowchart portion on different pages. One flow line enters or exits.
	Comment	Used to add comments or clarification.
	Magnetic disk storage	Used to show the storage in magnetic disk. More than one flow line can enter and exit

Guideline for flowchart :

-All necessary requirements should be stated in logical order.
-Should be clear, neat and easy to follow.
-Usually, the flow of direction is from left to right and top to bottom.
-If the flowchart becomes complex it is better to use the connector symbols to reduce the number of flow lines. We should avoid the intersection of the flow lines to make it more effective and better way of communication.
-Ensure that it has logical start and stop.
-Useful to test the validity of the solution, by passing through it with sample data.

Advantages :

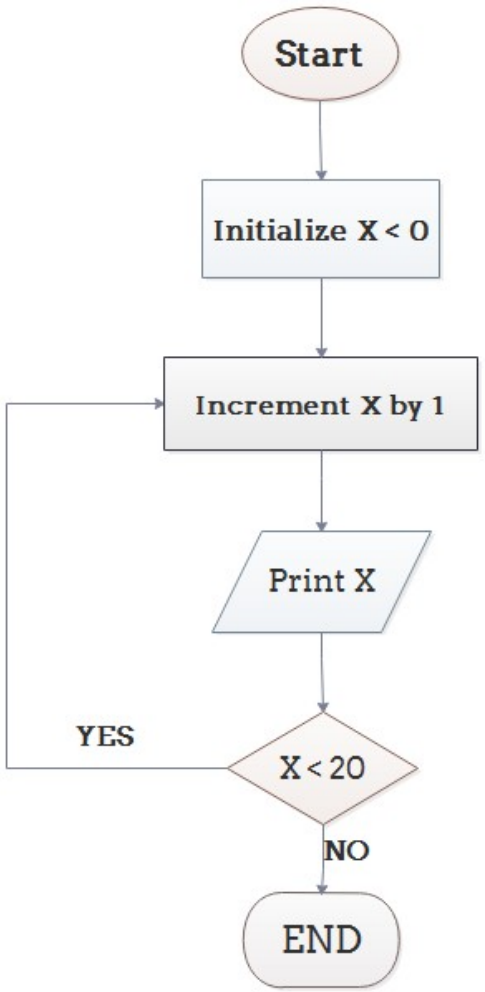
-Communication
-Effective analysis
-Proper documentation

-Efficient coding
-Proper debugging
-Efficient program maintenance

Limitation :
-complicated logic : for complicated logic it becomes complex and clumsy.
-Alteration and modifications may requires redrawing completely.

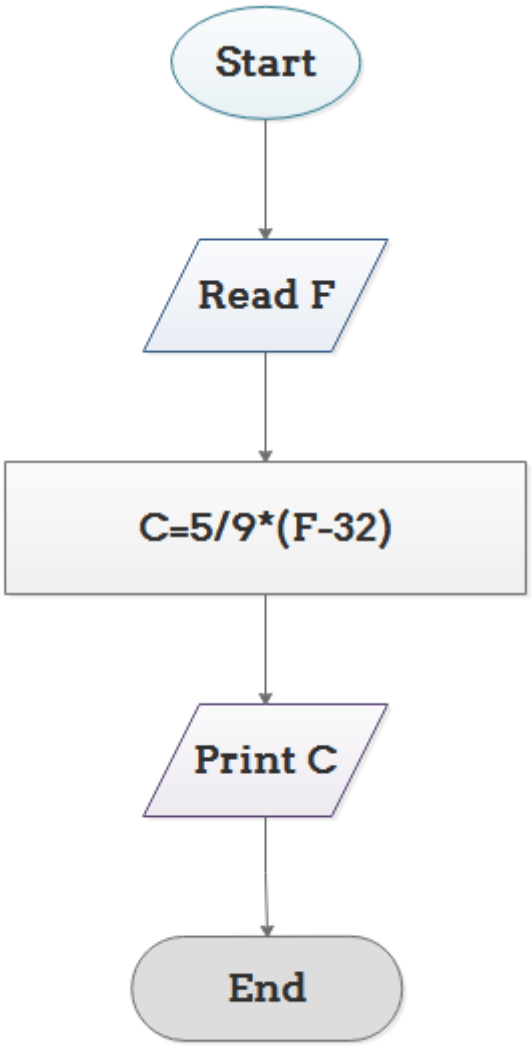
Examples……..
some examples:

Example 1: Print 1 to 20:
Algorithm:
Step 1: Initialize X as 0,
Step 2: Increment X by 1,
Step 3: Print X,
Step 4: If X is less than 20 then go back to step 2.
Flowchart:



Example 2: Convert Temperature from Fahrenheit (°F) to Celsius (°C)

Algorithm:
Step 1: Read temperature in Fahrenheit,
Step 2: Calculate temperature with formula C=5/9*(F-32),
Step 3: Print C,
Flowchart:



Types of Errors :

Sntax error :
-Any violation of the rules of the language.
Run time error :
-Errors such as mismatch of data type, referencing an out of bound array.
Logical error :
-Related to the logic of the program execution such as taking the wrong path,failure to consider the particular condition and incorrect order of evaluation of the statements belong to this category.
Latent errors :
-There are hidden errors that comes when particular set of data is used eg. divide by zero error.

Debugging and testing :

-Process of detecting and removing errors in a program, so that the program gives correct output.

i. **Debugging**

Process of isolating and correcting the errors.

Error isolation :

-Used to locate an error resulting in a diagnostic message

-We can find location of error by temporarily deleting the certain portion of the program and re running the program (deletion is accomplished by commenting the code).

Tracing :

-printf() statement is used to print the variables and state.

Watch values :

-It is value of variable or expression which is displayed continuously as the program executes. (Turbo C) Debug→watchs→add watch.

Break points :

-temporarily stopping point with in the program, used often in conjunction with watch values (Debug→add->Breakpoint)

Stepping :

-process of executing one statement at a time. Often used with watch values. (F7)

ii. **Testing**

-Execution of program in the intent of finding the errors.

-Testing is done to improve quality, for verification and validation, for reliability estimation.

#Human testing

-Effective error detection process done before computer based testing.

-It performs code inspection and review by the programmer or group

#Computer based testing

-Two step testing (compiler and run time testing)

Program Documentation :

-Keep information of all the phases while developing the software used for programmer as well as beginner. It should contain following.

1. A program analysis document with objectives, inputs, outputs and processing procedures.
2. Program design document algorithm and flowchart and other diagrams.
3. Program verification documents, with details of checking, testing and correcting procedures along with the test of the data.
4. Log is used to document the future program revision and maintenance activity.

Chapter-3
C Fundamentals

3.1 C tokens

The smallest individual units are called C tokens. C has six types of tokes.

Keywords	Constants	Operators
Identifiers	Strings	Special Symbols

3.2 Character set

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. Each **character** is represented by a number. The ASCII **character set**, for example, uses the numbers 0 through 127 to represent all English **characters** as well as special control **characters**. It can be grouped as:

1. Letters

   A-Z, a-z

2. Digits

   0-9

3. Special Characters

   , - comma
   . - period
   ; - semicolon
   : - colon
   ? - question mark
   ‘ - apostrophe
   “ - quotation mark
   ! - exclamation mark
   | - vertical bar
   / - slash
   \ - back slash
   ~ - tilde
   _ - underscore
   $ - dollar sign
   % - percent sign
   & - ampersand
   ^ - caret
   * - asterisk
   - - minus sign
   + - plus sign
   < - opening angle bracket(or less than sign)
   > - closing angle bracket(or greater than sign)
   (- left parenthesis
) - right parenthesis
   [- left bracket
   ] - right bracket
   { - left brace
   } - right brace
   # - number sign

4. White spaces

   -Blank Space

   -Horizontal Tab

   -Carriage Return

   -New Line

   Form Feed

ASCII CODE TABLE

DEC	HEX	OCT		DEC	HEX	OCT		DEC	HEX	OCT		DEC	HEX	OCT	
0			NUL	32			space	64			@	96			`
1			SOH	33			!	65			A	97			a
2			STX	34			“	66			B	98			b
3			ETX	35			#	67			C	99			c
4			EOT	36			$	68			D	100			d
5			ENQ	37			%	69			E	101			e
6			ACK	38			&	70			F	102			f
7			BEL	39			‘	71			G	103			g
8			BS	40			(	72			H	104			h
9			TAB	41			)	73			I	105			i
10			LF	42			*	74			J	106			j
11			VT	43			+	75			K	107			k
12			FF	44			,	76			L	108			l
13			CR	45			-	77			M	109			m
14			SO	46			.	78			N	110			n
15			SI	47			/	79			O	111			o
16			DLE	48			0	80			P	112			p
17			DC1	49			1	81			Q	113			q
18			DC2	50			2	82			R	114			r
19			DC3	51			3	83			S	115			s
20			DC4	52			4	84			T	116			t
21			NAK	53			5	85			U	117			u
22			SYN	54			6	86			V	118			v
23			ETB	55			7	87			W	119			w
24			CAN	56			8	88			X	120			x
25			EM	57			9	89			Y	121			y
26			SUB	58			:	90			Z	122			z
27			ESC	59			;	91			[	123			{
28			FS	60			<	92			\	124			|
29			GS	61			=	93			]	125			}
30			RS	62			>	94			^	126			~
31			US	63			?	95			_	127			DEL

3.3 Identifiers and keywords

Identifiers are names given to variables, constants, functions and arrays. These are user-defined names and consists of a sequence of letters and digits, with a letter as a first character. The underscore character is also permitted in identifiers, and it is usually used as a link between two words in a long identifiers.

Rules for Identifiers:

1. The first character of an identifier must be an alphabet or underscore.
2. Must consists of only letters(A-Z and a-z), digits (0-9) and underscore.
3. It can't use keywords.
4. It can't contain space.
5. Only first 31 characters are significant. (ANSI-C)
6. Identifiers are case sensitive. i.e. name and Name are two different identifiers.

Keywords are preserved words that has a special meaning in C language. The meanings can't be changed. Keywords serves as a basic building blocks of C program statements. The list of all keywords of ANSI C are listed below.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

3.4 Data Types

Data types is declaration for memory locations or variables that determines the characteristics of the data that may be stored and the methods(operations) of processing that are permitted involving them. C language is rich in data types. ANSI C supports three classes of data types.

1. **Primary (fundamental) data types**

The data type that are already defined in C. All C compiler supports five fundamental data types, namely integer(**int**), character (**char**), floating point(**float**), double-precision floating point (**double**) and **void**.

It can be described from the table below(**the following table is for 16-bit machine**).

Data Type	Keyword	Size (bits)	Range
Signed character	char, signed char	8	-128 to 127 (-2^7 to 2^7-1)
Unsigned character	unsigned char	8	0 to 255 (0 to 2^8-1)
Signed integer	int, signed int	16	-32768 to 32767 (-2^{15} to 2^{15}-1)
Unsigned Integer	unsigned int	16	0 to 65535 (0 to 2^{16} -1)
Signed Short Integer	short int, signed short int	8	-128 to 127
Unsigned short Integer	unsigned short int	8	0 to 255
Long Integer	long int	32	(-2^{31} to 2^{31} - 1)
Unsigned Long Integer	unsigned long int	32	0 to 2^{32}-1
Single Precision Floating	float	32	3.4E-38 to 3.4E+38
Double Precision Floating	double	64	1.7E-308 to 1.7E+308
Extended Double	long double	80	3.4E-4932 to 1.1E+4932

2. **Derived data types**

Data types that are derived from the fundamental data types. Example. Array, pointer structure and union.

3. **User-defined data types**

C supports the feature known as the “type defination” that allows the user to define an identifier that would represent an existing data type. For example.

typedef unsigned int ui;

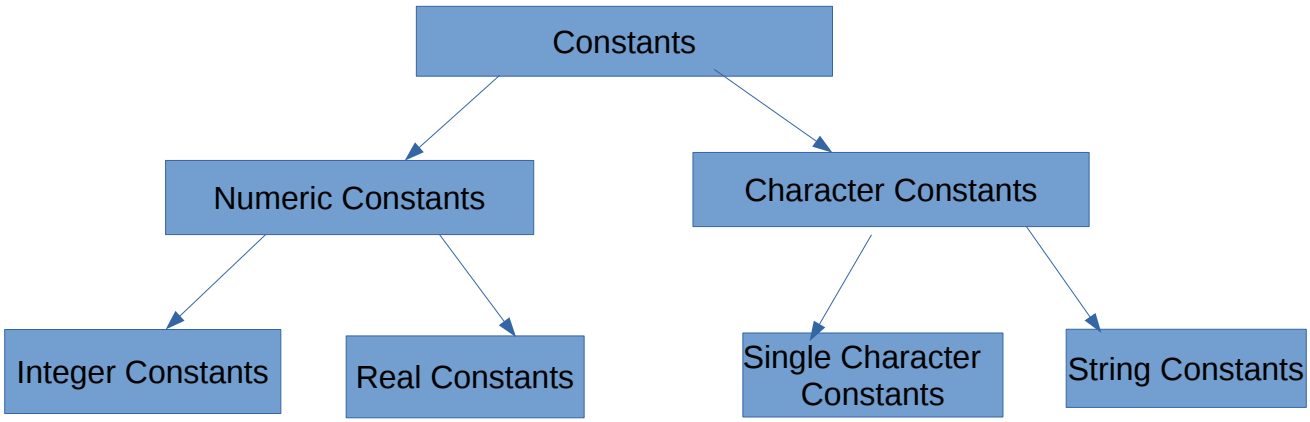
then unsigned int type data can be declared as.

ui ui_type;

Another user-defined data type is enumerated data type provided by the ANSI C. (enum)

3.5 Constants, Variables

Constants in C refer to fixed values that do not change during the execution of a program.



Integer Constants:

decimal integer :- 0 through 9, preceded by optional + and – example are -223, +243

hexadecimal integer :- A sequence of digits preceded by 0x or 0X, they contain 0 through 9, A through F or a through f. example are 0X2 0xF9

octal integer :- 0 through 7 with leading 0, example are 047, 0556

Real Constants:

These numbers are shown in decimal notation, having a whole number followed by the decimal point and the fractional part. Example are 215.90, 0.566, .99, +9.99, .98

Single Character Constants:

A single character constant contains single character enclosed with a pair of single quote marks. Example are '5', 'A', ';'

String Constants:

Sequence of characters enclosed in a double quotes. Example are "hello", "hello nepal 001"

Backslash Character Constant (escape sequence):

These represents single character, although they consists of two characters. These combinations are also known as escape sequence. They are used in output functions for example '\n' stands for newline character.

Constant	Meaning
'\a'	Audible alert (bell)
'\b'	Back space
'\f'	Form feed
'\n'	New line
'\r'	Carriage return
'\t'	Horizontal tab
'\v'	Vertical tab
'\''	Single quote
'\"'	Double quote
'\?'	Question mark
'\\'	backslash
'\0'	null

Variables:

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of the program, a variable may take different values at different times during execution.

Rules for naming variables(same as that of rules for naming **identifier**)

3.6 Declaration

Before designing and using suitable variable names, we must declare it.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

Primary Type declaration:

syntax:-
data-type var_name1, var_name2……., var_namen;
example:-
int my_int;

3.7 Pre-processor Directives

Before C program is compiled in a C compiler, source code is preprocessed by the program called preprocessor. This process is called preprocessing.
Commands used in preprocessor are called preprocessor directives and they begin with '#' symbol.
Commands used in preprocessor are called preprocessor directives.

Preprocessor	Syntax/ Description
Macro (An abbreviated name given is called macro)	#define This macro defines constant value that can be any of the basic data type.
Header file	#include <file_name_with_path>
Conditional compilation	#ifdef, #endif, #if, #else, #ifndef
Other directives	#undef, #pragma

3.8 Symbolic Constants

Symbolic constant is a sequence of character that substitute for a sequence of character that can't be changed. The character may represent a numeric constant, a character constant or a string constant. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. They are usually defined at the beginning of the program.
For example:
#define PI 3.1416, where PI is a symbolic constant whose value is 3.1416. On preprocessing all the occurrence of the PI is replaced by the 3.1416

Chapter-4
Operators and Expressions

4.1 Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually for the mathematical and logical expressions. The data and variables that operator operates are called operands.

4.2 Types of operators

Based on the operands they take:

1. Unary operator
   Operator that takes one operands. For example: pre-increment (++x), unary minus(-x)
2. Binary operator
   Operator that takes two operands. For example: Binary plus(x+y), logical and (x&y)
3. Ternary operator
   Operator that takes three operands. For example: (z = x>y?x:y)
   Also called conditional operator.

Based on the nature:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

4.2.1 Arithmetic operators

Operators that are used for all the arithmetic operations. They are listed below.

Operator	Meaning	Description
+	Addition or unary plus	
-	Subtraction or unary minus	
*	Multiplication	
/	Division	Integer division truncates the any fractional part. For example: x = 5/2, results 2 to be the value of x
%	Modulo division (remainder)	Modulo division can't be performed on floating point data. This operator gives the remainder. For example: x = 5%2, results 1 to be the value of x

4.2.2 Relational operators

Relational operators are basically used for the comparison purposes. By using relational operators, we often compare to quantities and take a decision based on the result. The result of the relational operator is either 1 (if true) or 0 (if false), that means 3 >2 results 1.

Operator	Meaning	Description
<	Is less than	
<=	Is less than or equal to	
>	Is greater than	
>=	Is greater than or equal to	

==	Is equal to	
!=	Is not equal to	

4.2.3 Logical operators

Logical operators are used when we want to test more than one condition and make a decisions. Logical expression also gives either 1 (if true) or 0 (if false).

Operator	Meaning	Description
&&	Logical AND	
||	Logical OR	
!	Logical NOT	

Truth table

operand-1	operand-2	operand-1 && operand-2	Operand-1 || operand-2
non-zero	non-zero	1	1
non-zero	0	0	1
0	non-zero	0	1
0	0	0	0

4.2.4 Assignment operators

Assignment operators are used to assign the result of an expression to a variable. Up to now we have used shorthand assignment operator "=". for example the statement **x = y+z**, the sum of the y and z are assigned to the variable x.
The another form of assignment operator is **v operator= expression;** where it is equivalent to
v = v operator expression;

for example, x += 3; means x = x + 3;

Expression with assignment operator	Detailed expression
x += y;	x = x + y
x -= y;	x = x - y
x /= y;	x = x / y
x %= y;	x = x % y
x &= y;	x = x & y
x |= y;	x = x | y
x ^= y;	x = x ^ y
x >>= y;	x = x >> y
x <<= y;	x = x << y

4.2.5 Increment and decrement operators

This operator is used to increment or decrement the original value of a variable. It is a unary operator. It itself can be categorized as **pre** and **post.**
When **postfix** is used (x++) with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
For example: let x = 10; y = x++; then the value of the y is 10 and that of x is 11

When **prefix** is used (++x) with a variable in an expression, the variable is incremented (or decremented) first and the expression is evaluated using the new value of the variable.
For example: let x = 10; y = ++x; then both x and y will have a value of 11.

4.2.6 Conditional operators

It is also called ternary operator. This operator takes three operands. The general form of the conditional operator is **exp1 ? Exp2 : exp3**; Where exp1, exp2, exp3 are all expressions.
Here the expression 1 is evaluated first, if the expression 1 results non-zero or 1, then the expression 2 is evaluated, else if expression 1 results zero, then the expression 3 is evaluated.

For example: program to find the largest of the two numbers.

```
#include<stdio.h>
int main()
{
   int a,b,c;
   printf("Enter the values of a and b");
   scanf("%d %d",&a,&b);
   c=(a>b)?a:b;
   printf("The Biggest Number is : %d ",c);
   return 0;
}
```


4.2.7 Bitwise operators

These operators are used for manipulation of the data at bit-level. It is not used for floating type datas.

Operator	Meaning	Explanation
&	Bit-wise AND	For example: let we have, x = 2; y =3; (In binary x = 0000 0010, y = 0000 0011) then, x & y = 2 (0000 0010) (representing only in 8-bits)
|	Bit-wise OR	For example: let we have, x = 2; y =3; (In binary x = 0000 0010, y = 0000 0011) then, x | y = 3 (0000 0011)
^	Bit-wise XOR	For example: let we have, x = 2; y =3; (In binary x = 0000 0010, y = 0000 0011) then, x ^ y = 1 (0000 0001)
<<	Left shift	For example: let we have, x = 2; y =3; (In binary x = 0000 0010)then, x<<2 = 8 (0000 1000)
>>	Right shift	For example: let we have, x = 2; y =3; (In binary x = 0000 0010)then, x>>1 = 1 (0000 0001)

4.2.8 Special operators

Some of the special operators available in C are.

operator	Meaning	Explanation
,	Comma operator	This can be used to link the related expressions together.
sizeof	Sizeof operator	A compile time operator that returns the number of bytes the operand occupies. For example: number_byte = sizeof(sum).
&	Reference	Gives the address of normal variable
*	Dereference	Gives the content of the pointer variable
.	Dot operator	Used to access the member of normal structure variable (More detail on chapter structure and union)
->	Arrow operator	Used to access the member of pointer structure variable (More detail on chapter structure and union)

4.3 Precedence and associativity

If more than one operators are involved in an expression, C has predefined rule of priority for the operators. This rule of priority of operators are called Precedence.
If the more than one operators of the same Precedence(priority) is present in an expression, then the order on which they execute is called the Associativity.
It can be shown in the table.

Operator	Description	Associativity	Rank(precedence)
( ) [ ]	Function call Array element reference	Left to Right	1
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Inrement Decrement Logical negation one's complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to Left	2
* / %	Multiplication Division Modulus	Left to Right	3
+ -	Addition (Binary plus) Subtraction (Binary minus)	Left to Right	4
<< >>	Left shift Right shift	Left to Right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right	6
== !=	Equality Inequality	Left to Right	7
&	Bit-wise AND	Left to Right	8
^	Bit-wise XOR	Left to Right	9
|	Bit-wise OR	Left to Right	10
&&	Logical AND	Left to Right	11
||	Logical OR	Left to Right	12
?:	Conditional Expression	Left to Right	13
= *= /= %= += -= &= ^= |= >>= <<=	Assignment operators	Right to Left	14
,	Comma operator	Left to Right	15

Chapter-5
Input and Output

5.1 Types of I/O

5.1.1 Unformatted I/O

Unformatted input and output functions are only work with character data type. Unformatted input and output functions do not require any format specifiers. Because they only work with character data type. Followings are the functions available for console I/O in C.

1)**getchar()**

This function reads the single character from the 'standard input' (keyboard). The getchar() function takes the following form.

```
char variable_name;
variable_name = getchar();
```


2)putchar()

This function writes a single character to the console. The general form of the putchar() function is

```
char ch = 'A';
putchar(ch);
```


3)gets()

This function reads full string even the blank spaces presents in a string. This function takes the following form.

```
char str[50];
gets(str);
```


4)puts()

This function prints the character array or string to the console window. This takes the following form.

```
char str[] = "hello hello";
puts(str);
```

Note:*similary, getche(), getch(), putch() functions are available.*

5.1.2 Formatted I/O

Formatted input/ output refers to and input and output data that has been arranged in a particular format. C provides scanf() and printf() functions for the formatted input and formatted output respectively. These functions accept, as parameters, a format specification string and a list of variables.
The format specification string is a character string that specifies the data types of each variable to be input or output and the size or width of the input and output.

#Formatted output / print() function.
The general form of the pritnf() function is:

printf("control string", arg1, arg2, ……. argn);

The control string consists of the following three types of items.

1. Characters that will be printed on the screen as they appear.
2. Format specification that define the output format for display of each item.
3. Escape sequence characters such as \n, \t, \b etc

A simple format specification has the following form.

% w.p type-specifier

The scanf() function returns the value of number of items that are successfully read. This value can be used to test whether any errors occurred during the reading.

I Output integer number

he format specification for printing and integer number is:

% w d

II Output of real numbers

The format specification for printing real number is :

% w.p f

We can display the real number in exponential notation by using the specificaion.

% w.p e

III Printing a single character

A single character can be displayed in the desired position using the format

%wc

IV Printing strings

The format specification is

% w.p s

Commonly used printf() format codes

Code	**Meaning**
%c	Print a single character
%d	Print a decimal integer
%e	Print a floating point value in exponent form
%f	Print a floating point value without exponent form
%g	Print a floating point value
%i	Print a signed decimal integer
%o	Print a octal integer, without leading zero
%s	Print a string
%u	Print a unsigned decimal integer
%x	Print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion character

h for short integers
l for long integers or double
L for long double

#Formatted input / scanf() function.

The function scanf() takes the following form.

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

The control string may contain

1. Field (or format) specifications, consisting of a conversion character %, a data type character (or specifier) and the oprional number, specifying the field width.
2. Blanks, tabs and new lines.

I)Inputing Integer number

The field specification for the reading an integer number is

% w sd

w is an integer number that specifies the field width of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode.

For example:

```
int var_x;
scanf("%4d", &var_x);
```

II)Inputting real numbers

unlike integer numbers the filed width of the real numbers is not to be specified and therefore simple **%f** specification is used.

For example:
float var_x;
scanf("%f", &var_x);

III)Inputting character strings
The specification for reading character string is as follows.

%ws or %wc

We have already seen that the single character can be read using unformatted I/O functions from the console. The scanf() function can be used to read single characters. In addition, the scanf() function can be used to read multiple characters.
For example:
(reading single character):
char ch;
scanf("%c", &ch);
(reading the multiple character):
char ch[10];
scanf("%5c", ch); or scanf("%s", ch);

Chapter-6
Control Statements

6.1 Decision making and branching statements

Followings are the statements for decision-making and branching in C

1. **if** statements
2. **switch** statements
3. **conditional operator** statements
4. **goto** statements

6.1.1 simple if statements

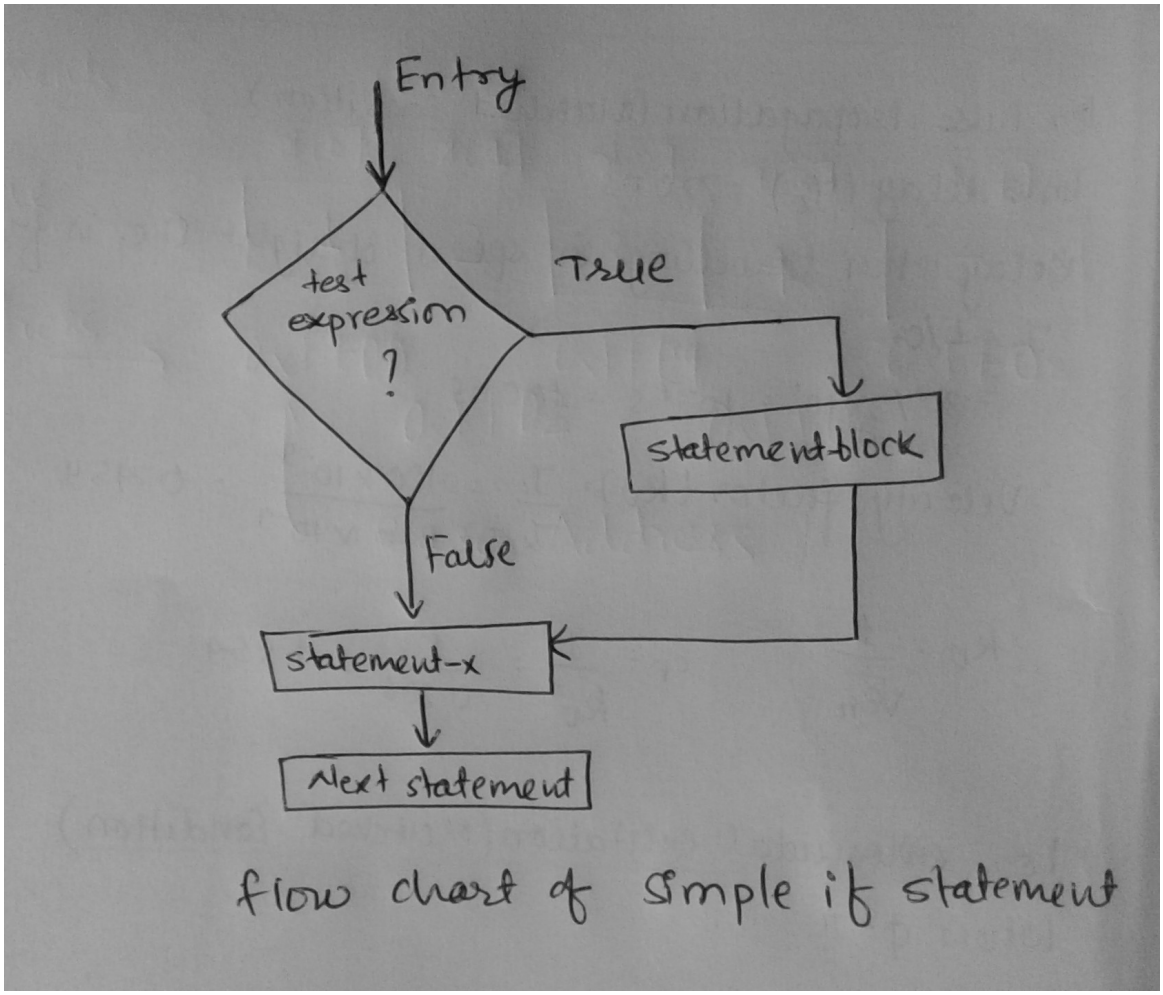
It is two way decision statement and is used in conjunction with expressions it takes the following form.

if(test_expression)

Syntax:

```
if (test_expression)
{
        statement-block;
}
statement-x;
```


The statement-block is either single statement or a group of statements. If the test_expression is true or non-zero then the statement-block is executed, otherwise the statement block is skipped and flow passes to the statement-x.



flow chart of simple if statement

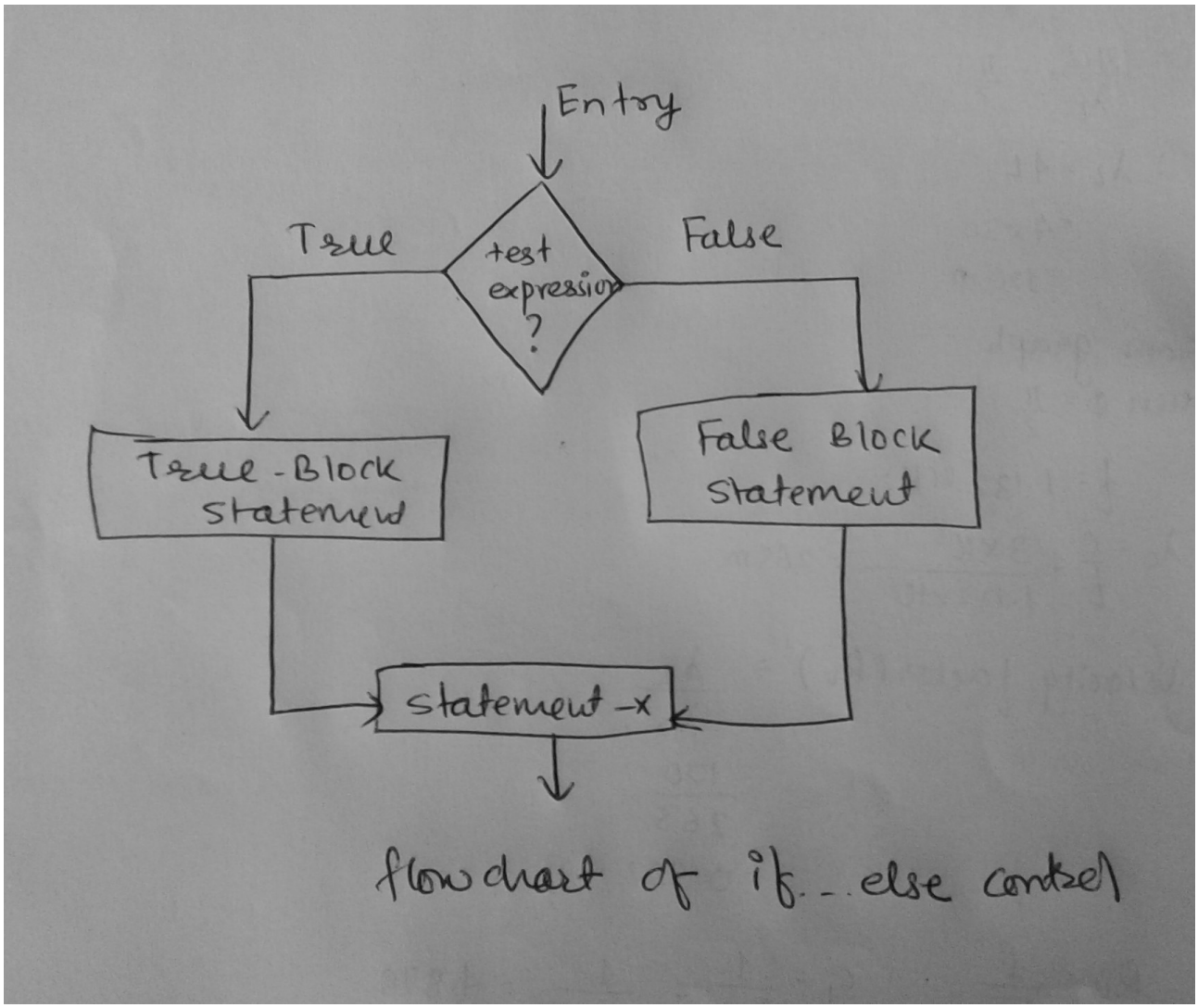
6.1.2 The if…..else statement

The if...else statement is the extension of the simple **if** statement.

Syntax:

```
if (test_expression)
{
        true-statement-block;
}
else
{
        false-statement-block;
}
statement-x;
```

If the test_expression is true, then the true-statement-block is executed, else the false-statement-block is executed.



6.1.3 Nesting of if….else statement

When a series of the decisions are involved, we have to use the more than one if… else statement in nested form.

Syntax:

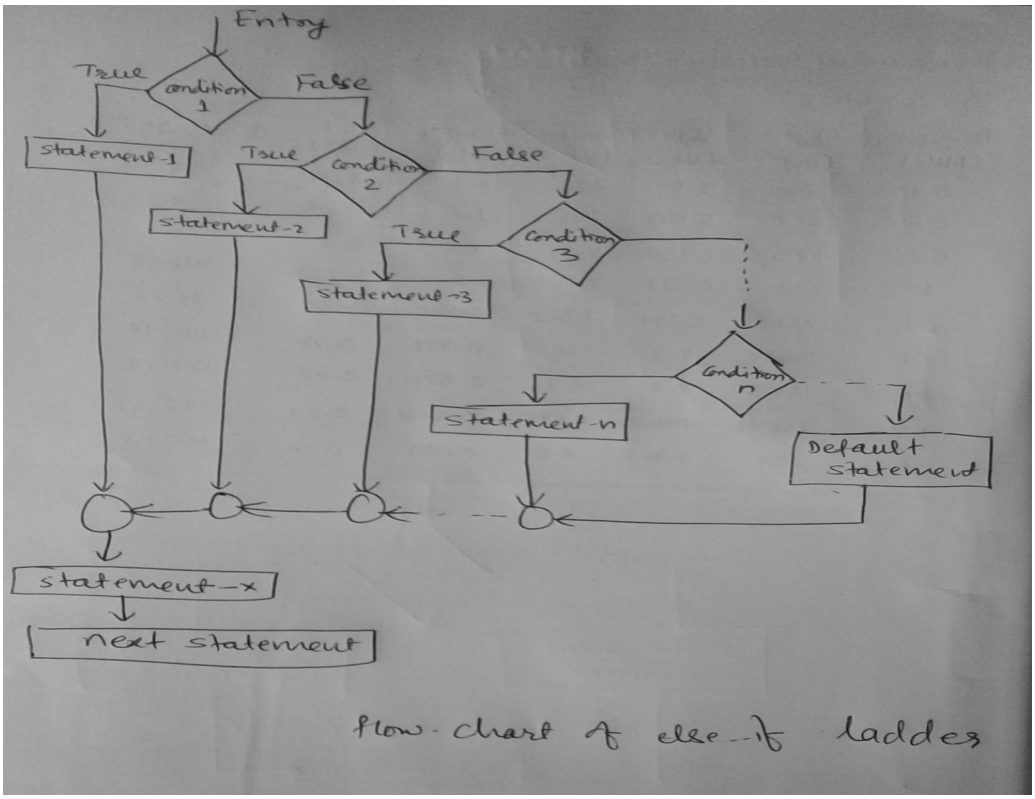
```
if (test_expression_1)
{
        if(test_expression_2)
        {
                inner-true-statement-block-1;
        }
        else
        {
                inner-false-statement-block-2;
        }
}
else
{
        outer-false-statement-block;
}
statement-x;
```

1.1.4 The else if ladder

When multipath decisions are involved the else if ladder can be used. The general construct is:

```
if(test_expression_1)
{
        statement-block-1;
}
else if(test_expression_2)
{
        statement-block-2;
}
else if(test_expression_3)
{
        statement-block-3;
}
.
.
.
.
else
{
        default-statement-block;
}
```

The conditions are evaluated from the top to downwards. As soon as the true condition is meet the statements associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else statement is executed.



6.1.5 The switch statement

The another alternative to the multipath decision is **switch** statement. As for many alternatives the else if ladder may be a bit awkward to use.

Syntax:

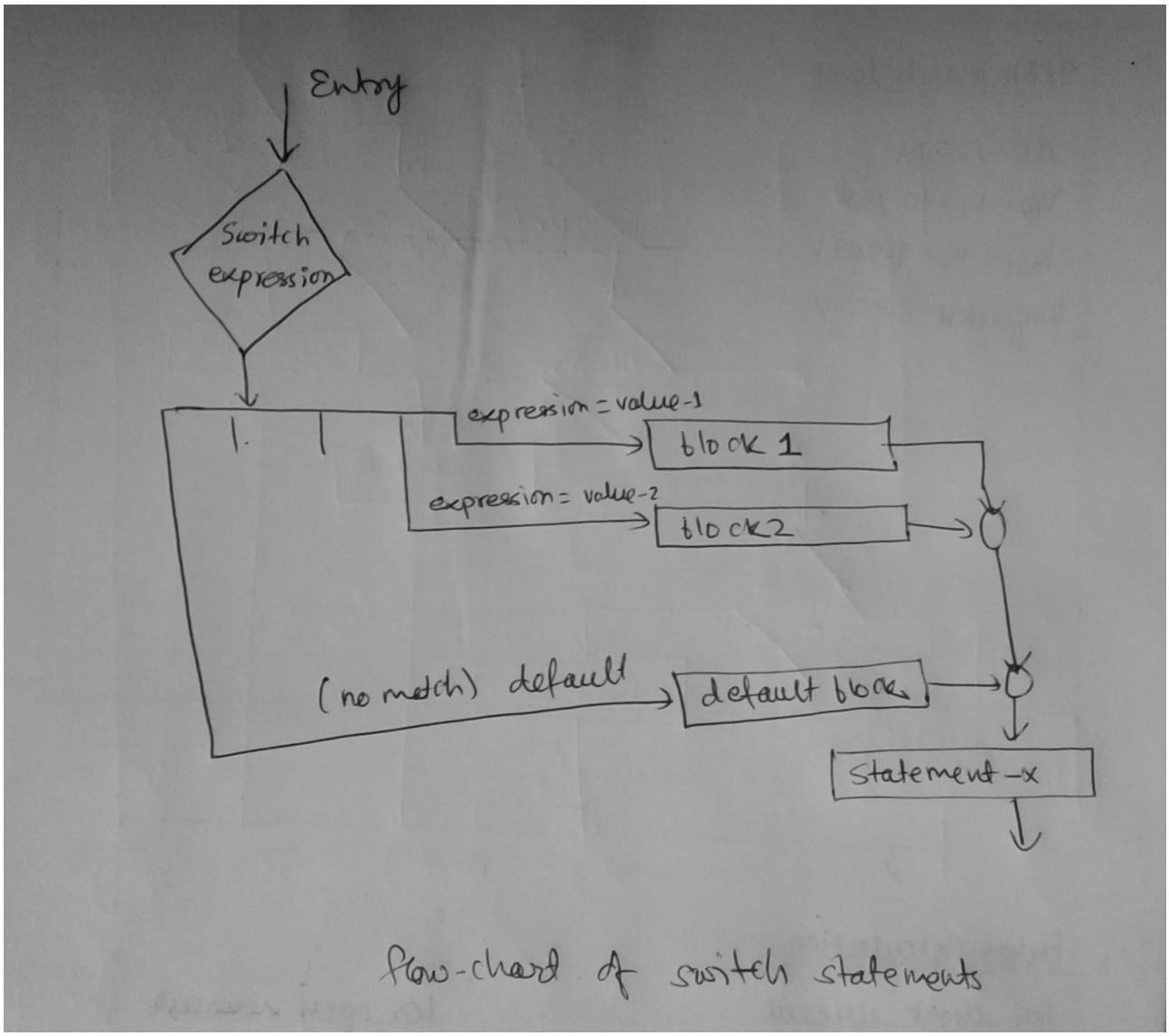
```
switch(test_expression)
{
        case value-1:
                block-1;
                break;
        case value-2:
                block-2;
                break;
        case value-3:
                block-3;
                break;
        ……
        ……
        case value-n:
                block-n;
                break;
        default:
                default-block;
                break;
}
statement-x;
```

The test_expression is an integer or character expression. And the value-1, value-2……. Value-n are constant expression evaluable to an integral constant known as case labels. And these values must be unique within a switch statement.

When the switch is executed, the value of the test_expression is compare to all the case values, and the matched case block statements are executed until the next break is reached. When none of the case matches then the default block statements are executed.

Rules for switch statement:

1. The switch expression must be an integral type.
2. Case labels must be constants or constant expression.
3. Case labels must be unique, No two case labels can have same value.
4. Case labels must end with colon.
5. The break statement transfers the control out of the switch statements.
6. The case label is optional. That means two or more case labels may belong to the same statement blocks.
7. The default label is optional, If not present no statements will be executed in case of no matched block.
8. There can be no more than one default label.
9. The default can be placed anywhere within the switch block, but generally placed at the end.
10. The switch statements can be nested.



flow-chart of switch statements

6.1.6 The goto statement

This statement is a unconditional branching. Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when to use of goto may be desirable.
The goto statement requires the label in order to identify the place where the branching is to be made. And the label is any valid identifier and must be followed by colon.
Synrax:
(forward jump)
goto label;
……
……
……
label:
statements-x;

(backward jump)
label:
statements;
……
……
……
goto label;

6.2 Decision making and looping statements

For the repeated execution of the statements C provides the following statements.

6.2.1 while statement

While is entry-controlled loop statement. That means the test_expression is evaluated first and there is no guarantee that the loop will be executed once.
Syntax:

```
while(test_expression)
{
        statement-of-while-block;
}
```

The test_condition is evaluated and if the condition is true, then the body of the while loop is executed. After the execution of the body of while body, the rest_condition is once again evaluated and if it is true, the body is executed once again, and the process is repeated until the test_expression is false.
The body of while statement can have one or more than one statements. If it contains only one statement then no braces are required. However it is good practice to use the braces even if the body has only one statement.

For example:

```
/* the following program prints the hello word ten times */

#include<stdio.h>

int main(){
        int i = 0;
        while(i < 10){
                printf("hello\n");
                i++;
        }
        return 0;
}
```

6.2.2 do….while statement

This is exit-controlled statement. That means the loop is executed at least once, and only then the test_expression is tested.
Syntax:

```
do{
```

```
        do-while-statement-blocks;
}while(test_expression);
```

The body of the do-while statement is executed and then the test_expression is evaluated, and if the test_expression is true then the body is again executed. The execution goes on until the test_expression is evaluated as false.

For example:
/* the following program reads the number until the user enters the negative number then find the sum of all the numbers and displays the sum*/

```
#include<stdio.h>

int main(){
        int I, sum = 0;
        do{
                printf("Enter the number:\n");
                scanf("%d", &i);
                sum += i;
        }while(i >=0);
        printf("sum of numbers = %d", sum);
        return 0;
}
```


6.2.3 The for statement
It is another entry-controlled loop that provides a more precise loop control structure. It has the following form.

```
for(initialization; test_condition; update_condition)
{
        body-statements-for-loop;
}
```

1. The initialization of the control variable is done first, using assignment statement.
2. The value of the control variable is tested in test_condition. The test_expression is any relational expression that when fails exits the loop.
3. On execution of the body-statements of the for loop, the test expression is evaluated and also the update_condition to change the control variable state.

For example:
/* Program to find the sum of all the even numbers between the given range*/

```
#include<stdio.h>

int main(){
        int lower, upper,i, sum= 0;
        prinf("Enter lower and upper range respectively:");
        scanf("%d %d", &lower, &upper);
        for(i = lower; i <=upper; i++){
                if(i % 2 == 0){
                        sum = sum + i;
                }
        }
        printf("sum = %d", sum);
        return 0;
}
```

Additional statements of for loop

1. More than one control variable can be initialized at once. For example: **for(i=0,j=0; i< 20; i++)**
2. Likewise the initialization the update section can have more than one part. For example: **for(i=0,j=0;i<10;i++,j++)**
3. The test_expression may have compound expression. For example: **for(i=0,j=0;i<10 &&j<20;i++,j++)**
4. The any of the part of the for loop can be empty. If all of the part of the for loop is empty then the loop is executed infinitely.

Nesting of the for loops
The for loops can be nested.
Syntax:

```
for(initialization_1; test_condition_1; update_condition_1)
{
        for(initialization_2; test_condition_2; update_condition_2){
                inner-for-block-statements;
        }
}
```


6.3 Jumping out of a loop
In the case, when we want to jump out of the loop or break the loop, C has a statement called break. Its general syntax is;

```
while(1)
{
        if(test_expression)
        {
                break;
        }
}
statement-x;
```


Here, the while statement is about to execute infinitely. The if evaluates the test_expression. If the test_expression is true or non-zero then the break statement is executed and the while loop is broken. The break statement can be used for any of the loops, while, do-while, for.

6.4 Skipping some part of the loop
C provides the statement continue to skip some part of the loop statements. Its general form is

```
while(1)
{
        ……..
        ……..
        statements-up….
        if(test_expression)
        {
                continue;
        }
        ……..
        ……
        statements-below…..
}
```

In above, if the test_expression is evaluated true or non-zero then the continue statement executes and the statements below the test_expression are skipped and the control is transferred to the top.

For example:
/* Program to find the sum of all the numbers between the given range except the number divisible by 5*/

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

```
        prinf("Enter lower and upper range respectively:");
        scanf("%d %d", &lower, &upper);
        for(i = lower; i <=upper; i++){
                if(i % 5 == 0){
                        continue;
                }else{
                        sum = sum + i;
                }
        }
        printf("sum = %d", sum);
        return 0;
}
```

Chapter-7
Functions

7.1. Introduction
Function is a group of statement to perform a specific task. Functions makes it easier for the programmer to make the code manageable and generic. With the help of functions, the large complex problems can be sub-divided into smaller components, that every components does some specific task.

7.2 Advantages of using functions
The functions has the following advantages.

1. The large complex problem/project can be subdivided into several independent tasks, and hence easier for the programmer to make the project manageable.
2. Functions helps to avoid the duplication of effort and code in the programs, making the program less memory intensive and easier to understand.
3. Functions helps to hide the implementation details of the program. That means, although we may not know how the functions work we can use the functions to accomplish specific task.
4. The division of the complex program/project into smaller components makes us easier to debug and test the problem individually.
5. The functions written once can be reused with little or no modification. This reduces the program development time and cost.

7.3 Types of function
Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C progrmming.

1. Standard Library Functions/ built-in functions
   The functions that are already available directly for use are called standard library funcitons. The **printf()** is a standard library function to send formatted output to the screen. This function is defined in **"stdio.h"** header file.
2. User-defined functions.
   The functions that are defined by the user, to solve the his/her specific task is called the user-defined functions.

7.4 Function prototype, definition and return statement
7.4.1 Function prototype/declaration
The function prototype/ declaration tells the compiler about the function name and return type(how the function is called). This helps the programmer to make the code manageable. By seeing the function prototype/ declaration the user/programmer knows how to use the function can be used. The function prototype can be declared as.
Syntax:

return_type function_name(argument_list);

For example:
If we want to make a function to add two numbers of integer type and returns the sum of integer type, then the function may look like.

int sum(int x, int y);

Here, the variable names x and y are optional for the prototype. That means the function may looks like.

int sum(int, int);

7.4.2 Function definition
Function definition gives the complete details about the function that how it works. The general syntax of the function definition is as follows.
Syntax:

```
return_type function_name(function_arguments){
        function-body-statements
}
```

For example:

The function two add two numbers as explained above can be defined as.

```
int sum(int x, int y)
{
        return x+y;
}
```

In function definition the names of the arguments x and y are compulsory as it was not compulsory in function declaration.
The function definition involves the following terms.

7.4.2.1 Return Type:

A function takes something, processes it and gives some result. The data type that the function gives or returns to the called location is called return type. The function on the other hand may not return any data, that means it may just processes or does some task and does not return any data. In such a case the return type of the function is mentioned as **void**.

7.4.2.2 Function name

This is the actual name of the function. The function name and the parameter list together constitute the function signature.

7.4.2.3 Parameters

Parameters are the optional in function. The function parameters are like placeholder. How many parameters does function takes depends on the type and the nature of the function that what it does.
The parameter names that is mentioned in the function definition are called **formal parameters**. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
The function_arguments mentioned above refers to the type, order and number of parameters of a function.

7.4.2.4 Function body

The function body contains the collection of the statements that function defines.

Categorization based on the return type and parameters.

1. Function with no argument and no return type.
   Example:
   void printHello();
2. Function with no arguments and return type.
   Example:
   f**loat returnPI();**
3. Function with arguments and no return type.
   Example:
   void printInt(int n);
4. Function with arguments and return type.
   Example:
   f**loat sum(float x,float y);**

7.4.3 Calling function

Until the function is not called, the function will not have any effect or existence. That means, to use the function and achieve specific task we have to call it.
When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached it return the program control back to the main program.
To call a function, you simply need to pass the required parameters along with the function name and the passed parameters are called **actual parameters**. If the function returns the value then you can store the returned value to somewhere.

Let's consider the above declared and defined example to find the sum of two variables.

```
int x = 10, y = 30, s;
s = sum(x, y);
```

Where, x and y are actual parameters of the function and the returned value is stored to the variable s.

7.5 Call by value and Call by reference

While calling a function there are two ways in which the arguments can be passed to the function.

7.5.1 Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case the changes made to the parameter inside the function have no effect on the argument. Let's consider the example of swapping of the two variable using pass by value and analyze why the swapping fails?

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

int main()
{
        int a = 30, b = 50;
        printf("Before swapping: A= %d, and B=%d\n", a,b);
        swap(a,b);
        printf("After swapping: A= %d, and B=%d\n", a,b);
        return 0;
}
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```


In this example, the values of the A and B from both the printf() statement gives the same result. Since, upon calling function swap(), the actual parameters a and b are actually copied to the formal parameters x and y. Logically, the swapping mechanism is correct inside the definition of swap() function. But, what really happened here was the swapping of the copied values of the a and b to x and y respectively. The two formal parameters x and y comes into existence upon calling the function swap() and dies when the function finishes. So, the actual swapping does not reflect on the actual parameters.

7.5.2 Call by Reference

This mechanism copies the address of the actual argument into the formal parameter. Inside the function since the address of the actual arguments are used, any changes made to the formal parameter get reflected to the actual parameters. The Call by reference can be achieved by using pointers(get details in pointer chapter).
Example:

```
#include <stdio.h>
void swap(int *x, int *y);
int main()
{
        int a = 30, b = 50;
        printf("Before swapping: A= %d, and B=%d\n", a,b);
        swap(&a, &b);
        printf("After swapping: A= %d, and B=%d\n", a,b);
        return 0;
}
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

The function parameters *x and *y indicates that the type of parameter is pointer type, that means it accepts pointer types variable. Since the function uses the address of the variable inside the function, any changes to the formal parameters get reflected to the original parameter.

7.6 Storage class (local, global, static, extern, auto, register)

7.6.1 automatic

The variable automatic, is local variable which is allocated and deallocated automatically when program flow enters and leaves the variable's scope. They are declared inside a function.
A variable declared inside a function without storage specification is by default, an automatic variable. The variable declared below is by default is automatic.

```
int main()
{
        int number;
        ……
        …….
}
```


We can declare automatic variable explicitly with the keyword **auto**.

```
int main()
{
        auto int number;
        ……..
        ……..
}
```


7.6.2 external

The keyword **extern** is used for the external variable declaration. The keyword **extern** tells the compiler that the variable is defined somewhere else in the program. The example below illustrates the concept.


```
int main()
{
        extern int number; /* external declaration */
        ……
        ……..
}
function_1()
{
        extern int number; /* external declaration */
        ……
        ……...
}
int number;
```

In above, the variable **number** is declared at last of the **main()** and **function_1()** function. The variables used inside those functions refer to the same variable, since they are declared as external inside both of the functions.

7.6.3 static

The value of the static variable persists until the end of the program. The variable can be declared static using the keyword **static**. The static variable can be local or global depending on the location of the declaration. The static variable born or comes into existence once the scope of the variable is invoked and persists till the program end. The static variable is initialized once.

Example:

```
void counter();

int main()
```

```
{
        int i;
        for(i = 0; i < 5; i++)
        {
                counter();
        }
        return 0;
}
void counter()
{
        static int count = 0;
        printf("count=%d\n", count);
        count++;
}
```

Expected Output:
count=0
count=1
count=2
count=3
count=4

In case of the variable not declared as static the output would have like
count=0
count=0
count=0
count=0
count=0

It is because, In every call of the function counter(), the variable count is initialize to zero, and hence resulting 0 to every output.

7.6.4 register

For the faster operation or calculation of the variable, it can be kept directly to the processor's register. And the keyword **register** can be used. Since the number of registers available in any of the processor is limited, the declaration of the variable as register is just the request to the compiler. That means there is no guarantee that the variable declared as register will be register variable. The general syntax is:

register int x;

7.6.5 local

Function that are declared inside the function or block are called the local variables. They can be used or known only inside the function where they are declared. For example, the variables a and b are local to the main function,

```
#include <stdio.h>
int main()
{
        int a, b;
        ……..
        ……..
        return 0;
}
```

7.6.6 global

The variables that are declared outside the function , generally on top of the program are global variables. They hold their value or live throughout the lifetime of the program. They can be accessed inside any of the functions. For example the variable glob_var below is a global variable.

```
#include <stdio.h>
int glob_var;
int main()
{
        int a, b;
        ……..
        ……..
        return 0;
}
```

7.6 Recursive function

A function that calls itself is called recursive function and the process is called recursion. While using recursion the programmer needs to be very careful to define the exit condition for the function, otherwise it will go in an infinite loop. General syntax:


```
#include <stdio.h>
void recursion(){
        recursion(); /* function calls itself */
}

int main()
{
        recursion();
}
```


For example:


```
/* program to calculate the factorial of a number using recursion */
#include <stdio.h>
unsigned long int factorial(unsigned int i){
        if(i <= 1){
                return 1;
        }
        else
        {
                return i * factorial(i – 1);
        }
}

int main()
{
        int number = 10;
        printf(“The factorial of %d = %d\n”, number, factorial(number));
        return 0;
}
```

Recursion versus iteration

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

Chapter-8
Arrays and strings

8.1. Introduction

Array belongs to a derived data type. Array is a fixed-size sequenced collection of elements of the same data type. It is simply a grouping of like-type data. We can use arrays to represent not only simple list of values but also tables of data in two three or more dimensions.

1. One-dimensional array
2. Two-dimensional array
3. Multidimensional array

An array share a common name for all the elements. The individual elements of an array can be accessed using subscript with the name. The array elements are stored in memory in a contiguous fashion.

8.2. Single and multi dimensional array

8.2.1 One dimensional array.

A list of items can be given a one variable name using only one subscript and such a variable is called single-subscripted variable or a one-dimensional array.
Syntax to declare single dimensional array.
data_type variable_name[SIZE];

For example:
int my_array[20];
The array represents the 20 elements of integer type. The element subscript starts from 0 and goes up to 19, constructing total of 20 array elements. The first element of an array is my_array[0] and that of last is my_array[19].

Initialization of one dimensional array.
(Compile time/ Inline initialization)
Syntax:
data_type variable_name[SIZE] = {list_of_values};

The memory size that the array holds is the **SIZE * size_of_the_data_type;**
For example:
int number[3] = { 4, 5, 6}; or int number[ ] = { 4, 5, 6}; /* size is optional for inline initialization */

Another valid examples are.
int number[3] = {4}; /* here the first element is initialized to 4 and rest others to zero */
int number[2] = {4, 5, 6, 7}; /* this is illegal more than the size is not valid */

Next important thing is that, the corresponding elements are initialized to corresponding index elements.

8.2.2 Two dimensional array.

An array of array is called two dimensional array. The data in two dimensional array is constructed like matrix elements. They can be expressed or represented in the form of the rows and columns. The two dimensional array uses the two subscript for the notation.
The general syntax for the declaration of two dimensional array is:
data_type array_name[ROW_SIZE] [COL_SIZE];

The memory size that the array holds is the **ROW_SIZE * COL_SIZE * size_of_the_data_type;**
Each dimension of an array is indexed from zero to maximum size minus one.
For example:

int int_array[3][3];

The above statement declares the 3x3 two dimensional array of integer type. These elements can be more clearly represented in diagram as:

int_array[0][0]	**int_array[0][1]**	**int_array[0][2]**
int_array[1][0]	**int_array[1][1]**	**int_array[1][2]**
int_array[2][0]	**int_array[2][1]**	**int_array[2][2]**

Initialization:
Two dimensional array can be initialized as:
data_type array_name[r_sieze][c_size] = {list_of_elements}; or
data_type array_name[r_sieze][c_size] = {{first_row_list},{second_row_list},{third_row_list}……..};

During inline initialization, of the array the first dimension is compulsory, and the another one is optional. That means the above statement can be like.
data_type array_name[ ][c_size] = {{first_row_list},{second_row_list},{third_row_list}……..};

8.2.3 Multidimensional array
C allows the array of multidimensional, The multidimensional array can be declared as:

data_type array_name[size_1][size_2][size_3]……..[size_n];

8.3. Passing array to functions

Passing One dimensional array:
Like passing the values of simple variable to a function, it is also possible to pass the values of an array to a function. To pass one-dimensional array to a function it is sufficient just by mentioning the name of an array without any subscript.
The calling of such function may look like:
function_name(array_name);

The function header/prototype may look like:
data_type function_name(data_type array_name[]);

Here, the array subscript is compulsory but not the size.
The passing of array to a function is pass by reference. That is, any changes to the array elements inside the function is reflected to the original array elements.

Rules to pass an array to a function:
1. The function must be called by passing only the name of an array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

Let's consider the simple example for passing one-dimensional array to a function.

```
/* PROGRAM TO FIND THE LARGEST OF THE FLOATING
   TYPE ONE DIMENSIONAL ARRAY              */

#include <stdio.h>
#define N 10
/* function header */
float largest(float arr[], int size);

int main()
{
   int i;
```

```
    float array_elements[N], lar;
    /* reading array elements */
    printf("Enter array elements:");
    for(i = 0; i < N; i++){
        scanf(" %f", &array_elements[i]);
    }
    lar = largest(array_elements, N);
    printf("\nLargest of an array = %f", lar);
    return 0;
}
/* function defination */
float largest(float arr[], int size){
    float max;
    int i;
    max = arr[0];
    for(i = 0;i < size; i++){
        if(max < arr[i]){
            max = arr[i];
        }
    }
    return max;
}
```


Passing Two dimensional array:
The passing of two dimensional array to a function is similar like one dimensional array. The simple rules for two dimensional array are:

1. The function must be called by passing the array name
2. In the function definition, we must indicate that the array has two dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration must be same like the function definition.

8.4. Strings

The null-terminated character array is called string. Any group of characters defined between the double quotation mark is called string constant. The string is actually a character array, it can be declared as:
char array_name[size];
The size dimension is the number of character in the string. When a compiler assigns the character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore the size should be maximum number of the character in the string plus one.
The initialization of the character array or string can be done as:
char name[15] = "Hari Bahadur"; /* compiler automatically assigns null character */
char name[15] = {'H', 'a', 'r', 'i', ' ', 'B', 'a', 'h', 'a', 'd', 'u', 'r', '\0' };
/* we must list null character as well for this style */

Reading the string from the terminal:
The **scanf()** function can be used with %s specification to read the string. For example:
char address[10];
scanf("%s", address);
Since the address itself is the address of the first character element of the address array, the & is not required here for the scanf() function.
The above mentioned mechanism does avoid reading of the characters after the first occurrence of the space.

For example:

```
/* program to check if the string is pallindrome or not */
#include <stdio.h>
int main(){
	char my_string[40];
	int state = 1, i, length = 0;
	printf("Enter the string to check:");
	scanf("%[^\n]", my_string);
	/* calculating length */
	for(i = 0; my_string[i] != '\0'; i++){
		length++;
	}

	/* checking pallindrome */
	for(i = 0; i <= length/2; i++){
		if(my_string[i] != my_string[length - i -1]){
			state = 0;
			break;
		}
	}

	if(state == 1){
		printf("Entered string is PALLINDROME.\n");
	}else{
		printf("Entered string is NOT PALLINDROME.\n");
	}

	return 0;
}
```


8.5. String Handling functions

C provides library “string.h” header file for the various kinds of string manipulations. Some of the string handling functions are described below.

1. **strcat()**
   This function joins two string together. It takes the following form.
   strcat(string1, string2);
   Where the string1 and string2 are character array. The string2 is appended to the string1 by removing its null character.
2. **strcmp()**

This function compares the two strings. This function returns zero if they are equal and numeric difference if they are not. It takes the following form.

strcmp(string1, string2);

3. **strcpy()**
   This function copies the one string to another. This takes the following form.
   strcpy(destination_string, source_string);
   The source_string is copied to the destination_string.

4. **strlen()**
   This function counts and returns the number of character in the string. Its general form is:
   n = strlen(string);
   Here, the length of the string string is returned to the integer variable n.

5. **strncpy()**
   This works same like **strcpy()** but, just copies the n-leftmost characters of the source_string.

6. **strncmp()**
   This is just the variation of the strcmp(), It compares only the first n characters of the strings.
7. **strncat()**
   Also the variation of the strcat(). It just copies the n left-most character of the source_string to the end of the destination string.
8. **strstr()**
   This is a two-parameter function that can be used to locate a sub-string in a string. It takes the following form.

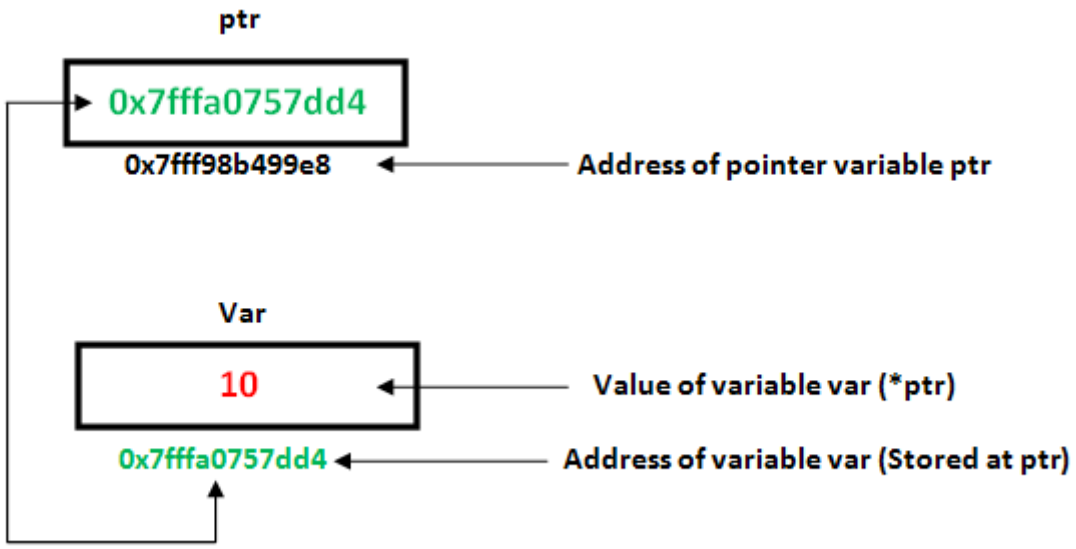
strstr(str1, “ab”);

Chapter-9
Pointers

9.1. Introduction

Pointers are derived data type in C. They contain memory addresses as their values. Since they store memory addresses, they can be used to to access and manipulate data stored in the memory. The benefits of the pointers are.

1. Pointers are the more efficient in handling arrays and the data tables.
2. Pointers can be used to return the multiple values from the function via function arguments.
3. Pointer return references to functions and thereby facilitating passing functions as arguments to other functions.
4. The use of pointer array to character strings results in saving the data storage space in memory.
5. Pointer allow C to support dynamic memory allocation.
6. Pointers provide efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks, and trees.
7. Pointers reduce length and the complexity of the programs.
8. They reduce the execution speed and thus reduce the program execution time.



Pointer Constant:
Memory addresses within a computer are referred to as pointer constants. We can not change them we can use them to store the data values.

Pointer Values:
We can not save the value of memory addresses directly. We can obtain the value through the variable stored there using address/ reference (&) operator. The value thus obtained is called pointer value.

Pointer variable:
The variable that contains or store the pointer value is called pointer variable.

9.2 Pointer Declaration/Initialization

The declaration of the pointer variable takes the following form.

data_type *pt_name;

It gives the information about.

1. The asterisk(*) tells the compiler that the variable pt_name is a pointer variable.

2. pt_name needs memory location.
3. pt_name points to a variable of type data_type.

For example:
float *f_ptr;
Here, the f_ptr is a pointer variable of type float. That means it stores the address of the floating type variable.
Initialization:
The pointer variable needs to be assigned the address of the another variable, and the process of the assigning the address of another variable to a pointer variable is known as initialization. It can be done as:

```
int normal_variable;
int *pointer_varibale; /* pointer variable declaration */
pointer_variable = &normal_variable; /* pointer variable initialization */
```

In above, the pointer variable needs to be compatible. The following is wrong.

```
float normal_variable;
int *pointer_varibale;
pointer_variable = &normal_variable; /* WRONG because of incompatible type initialization */
```

The pointer variable can't be initialized with arbitrary number.

```
int *pointer_varibale = 98979;  /* WRONG */
```


Accessing a variable through its pointer:
Once the pointer variable is initialized, then we can access the value of variable using the pointer varibale. The mechanism is accomplished using indirection/dereference operator (*). This is shown below.


```
int normal_variable, *p, n;
normal_variable = 120;
p = &normal_variable;
n = *p; /* here the value of the variable is accessed using the pointer variable  */
```


Chain of Pointers:
It is possible to make a pointer to point another pointer, thus creating the chain of pointers. It can be declared as: **data_type **ptr;**
Here, it means the ptr can store the address of another pointer variable. It can be shown as

```
int x, *p1, **p2;
x = 100;
p1 = &x; /* p1 takes the address of x */
p2 = &p1; /* p2 takes the address of p1 */
```

9.3 Pointer Arithmetic

The pointer variables must follow the rules mentioned below.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be assigned/initialized with NULL or 0 value.
4. The pointer variable can be pre-fixed or post-fixed with increment and decrement operators.
5. An integer value can be added or subtracted from the pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another pointer variable.
7. When two pointers point to the same data type, they can be compared using relational operators.
8. A pointer variable can not be multiplied by a constant.
9. Two pointer variable can not be added.
10. An arbitrary can not be assigned to a pointer variable.

9.4 Passing pointers to a function

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The processes of calling a function using pointers to pass the addresses of a variables is known as call by reference. For example:

```
#include <stdio.h>
void increaseThree(int *val)
{
        *val += 3;
}
int main()
{
        int i = 10;
        increaseThree(&i); /* call by reference or address */
        return 0;
}
```

As mentioned earlier, the pointers can be used to return multiple values. The process involves the passing of multiple arguments as pointer.

9.5 Function returning pointers

Similar like returning the values from a function, a function can also return pointer. Let's consider the example:

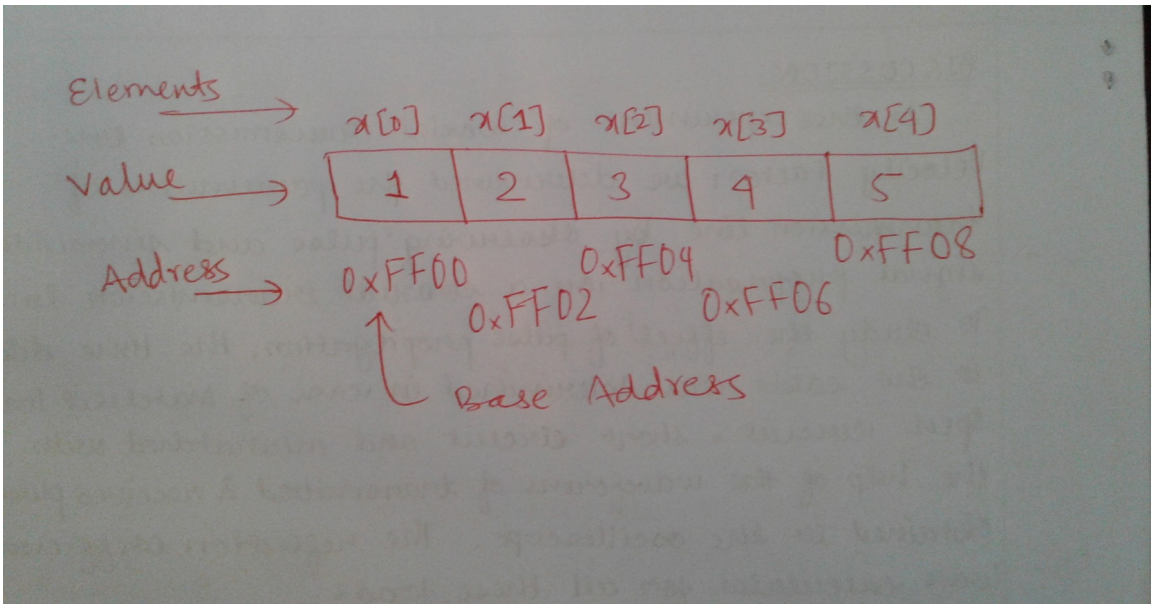

```
/* Program to find the largest of the two elements */

#include <stdio.h>
int * largest(int *a, int *b){
   if(*a > *b){
      return a;
   }else{
      return b;
   }
}

int main()
{
   int x = 10, y = 50;
   int *lar;
   lar = largest(&x, &y);
   printf("Largest of two is : %d", *lar);
   return 0;
}
```


9.6 Relationship between arrays and pointers

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address or array name is the location of the first element (index 0) of the array. The array name is the **constant pointer** to the first element.

Let's consider an array as shown:
int x[5] = {1, 2, 3, 4, 5};

Suppose the base address of x is 0xFF00, now the five elements will be stored as



The name x is defined as the constant pointer pointing to the first element, x[0] and therefore the value of x is 0xFF00, is the location where x[0] is stored.

X = &x[0] = 0xFF00

Suppose let we declare the pointer variable **ptr** that points to the integer data type.

int *ptr;

ptr = x; or ptr = &x[0];

Then the relationship between ptr and x can be shown as(for address):

ptr = &x[0] = 0xFF00
ptr + 1 = &x[1] = 0xFF02
ptr + 2 = &x[2] = 0xFF04
ptr + 3 = &x[3] = 0xFF06
ptr + 4 = &x[4] = 0xFF08

Similarly for values:

***ptr = x[0] = 1**
***(ptr + 1) = x[1] = 2**
***(ptr + 2) = x[2] = 3**
***(ptr + 3) = x[3] = 4**
***(ptr + 4) = x[4] = 5**

Note: Please search for the relation between pointer and two dimensional array.

9.7 Array of pointers

Like, array of normal variable the is also possible to declare array of pointers. One of the important use of it is handling of a table of strings. Let's consider the simple example with the declaration of array of pointer with initialization

char *name[3] = {"ram", "shyam", "shivashankar"};

Here, the element name[0] points to the "ram", and similarly others.

9.8 Dynamic Memory Allocation(DMA)

The process of allocating memory at run time is known as dynamic memory allocation. Meaning that the memory that pointer variable takes is known at run time only. There are four library routines known as “memory management functions” that can be used for allocating and freeing memory during program execution.

1. **malloc()**

A block of memory may be allocated using the function malloc(). The malloc() function reserves a block of memory of specified size and returns a pointer of type void. This means that the returned pointer can be assigned to any type of pointer. It takes the following form.

ptr = (cast-type *) malloc(byte_size);

Let’s consider an example allocating memory for twenty integer elements.

Int *i_ptr;
i_ptr = (int *) malloc(20 * sizeof(int));

2. **calloc()**

calloc() allocates the multiple block of storage, each of the same size, and sets all bytes to zero. And pointer to the first byte of the allocated region is returned. If there is not enough space then NULL pointer is returned.

Let’s consider the example of allocating the 30 structure variable of type student.(refer structure chapter for structure)

```
…….
…….
struct student
{
        char name[20];
        int roll;
        float marks;
};
struct student *record_ptr;
record_ptr = (struct student *) calloc(30, siezeof(struct student));
……..
……..
```

3. **free()**

The compile-time storage of a variable is allocated and released by the system based on the scope/storage class. The dynamically created variables needs to be released when the variable is no longer necessary. The **free()** function does so. The general syntax is:

free(pointer_to_be_released);

4. **realloc()**

In case of insufficiency or changes to the previously allocated memory, then it is possible to change the allocation size. The function realloc() does so. Its general syntax is:

ptr = (type_cast *)malloc(old_size);
ptr = (type_cast *)realloc(ptr, new_size);

```
/* example of malloc and free */
#include <stdio.h>
#include <stdlib.h>

int main()
{
	float *arr_ptr, sum = 0;
	int n,i;
	/* here we use dynamic memory allocation
		you can use array for fixed number of
		array size
	*/
	printf("Enter the number of elements of an array:");
	scanf("%d", &n);
	arr_ptr = (float *)malloc(sizeof(float) * n);

	printf("Enter array elements:");
	/* reading array elements */
	for(i = 0; i < n; i++){
		scanf("%f", arr_ptr + i);
		sum += *(arr_ptr + i);
	}
	printf("\nSUM = %f\n", sum);
	free(arr_ptr);
	return 0;
}
```

Chapter-10
Structure and Unions

10.1 Structure Introduction

Structure is a convenient tool for handling a group of logically related data items. That means structure is collection of homogeneous data items. Structure helps to organize complex data in a more meaningful way. In real life, when creating a software/ programs we have to model the entities in real existence in the form of data. For example, what if we want to represent student in program. Actually the student data can be constructed by combination of attributes like name, address, roll_number, marks etc. Hence, the individual data like name of character string, address of character string, roll_number of int type and marks of float type, can be combined together to form a complete student data. And such a mechanism is facilitated by the structure in C.

10.1.1 Defining a structure

For the structure variable to be created or declared. We must first define the structure format. Hence the defining the structure gives us a template for the data type. No memory is allocated on structure definition. The compiler just takes the memory for the information about the template.
Syntax for defining structure:

```
struct struct_name
{
        data_type1 member_1;
        data_type2 member_2;
        ……..
        ……..
        ……..
};
```


For example let's define structure for student:

```
struct student
{
        char name[30];
        int roll;
        float marks;
        char address[30];
};
```

Rules defining the structure:

1. The template must be terminated by semi-colon.
2. struct_name such as student is used later for the variable declaration.
3. The attributes inside the structure defination is considered as individual defination so, they are declared on seperate lines though they can be stated in same line.

10.1.2 Declaring the structure variable

The structure variable can be declared same like normal variable declaration. Since the structure definition makes available the user defined type data, when declaring variable **struct** keyword and **structure_name** must be supplied. The general syntax is:

```
struct structure_name variable_name;
```

For example:
from the above defined structure.

```
struct student st1;
```

The structure variable can also be declared, along with definition, and this can be also declared in two ways.

```
struct student
{
        char name[30];
```

```
        int roll;
        float marks;
        char address[30];
} st1, st2; /* declares the two structure variables st1 and st2  */
```
Later on, we can declare other variables as per the requirement.

```
struct
{
        char name[30];
        int roll;
        float marks;
        char address[30];
} st1, st2; /* declares the two structure variables st1 and st2  */
```
Later on the structure variable can not be declared.

typedef defined structure
We can use the keyword typedef to define a structure as follows.
```
typedef struct
{
        char name[30];
        int roll;
        float marks;
        char address[30];
} STUDENT;
```
Then the variable can be declared as:
STUDENT st1; /* which is more convenient */

10.1.3 Accessing structure members
The structure variables has its members, the members of the structure variable can be accessed following ways.

10.1.3.1 dot operator/ period operator
For normal structure variable this operator is used to access the members of the variable. Lets consider the above example:
```
struct student
{
        char name[30];
        int roll;
        float marks;
        char address[30];
};
struct student s1;
s1.name; /* accessing the name of the student */
s1.roll; /* accessing the roll of the student */
scanf(" %s", s1.name); /* reading the name of the student */
scanf(" %d", &s1.roll); /* reading the roll of the student */
```

10.1.3.2 arrow operator
If the structure variable is of type pointer then the members of the structure can be accessed by → arrow operator. Lets consider example:
```
struct student
{
        char name[30];
        int roll;
        float marks;
        char address[30];
};
struct student s1, *sp;
```

```
sp  = &s1;
sp->name; /* accessing the name of the student */
sp->roll; /* accessing the roll of the student */
scanf(" %s", sp->name); /* reading the name of the student */
scanf(" %d", &sp->roll); /* reading the roll of the student */
```

10.1.4 structure initialization
The structure variable can be initialized like:

```
struct student s1;
s1 = {"hari", 30, 89.99, "kathmandu"};
```

Here, the order of members must match. It is permitted to have partial initialization, we can initialize only the first few members of the structure. The uninitialized members will be assigned default values, Zero for integer and floating point variables and '\0' for the character and strings.

10.2 Structure within a structure

The structure itself can contain structure inside it. This can be illustrated as:

```
struct student
{
        char name[30];
        int roll;
        struct
        {
                float sub1;
                float sub2;
                float sub3;
        } marks;
        char address[30];
};
```


In above example if we want to access the sub1 marks then.

```
struct student s1;
s1.marks.sub1; /* accessing the sub1 marks */
```

10.3 Array Of Structure

Similar like array of other normal variable, it is also possible to create the array of structure variable. Let's create the structure variable of student.

```
struct student s[20]; /* creates the structure array of 20 of student type */
```

s[14].name; /* access to the name of 15th element of an array*/

10.4 Structure and pointers

The structure variable same like normal variable, is possible to create pointer. For example.

```
struct student
{
        char name[30];
        int roll;
        float marks;
        char address[30];
};
struct student *sp; /* sp is pointer variable of type struct student */
```

See example programs for more details/uses of the pointer.

10.5 Passing structures to a function

Similar like normal variable the structure variable can be passed to functions. *See example programs for more details.*

/* Example Program */

Note: capital letters are considered to be come first

```
#include <stdio.h>
#define N 8

struct employee{
        char name[30];
        char address[30];
        float salary;
};
void read(struct employee *e);
void display(struct employee e);
int find_length(char *str);
int compare_string(char *str1, char *str2);
void sort(struct employee e[], int size);
int main()
{
        struct employee emp[N];
        int i;
        /* reading employee */
        for(i = 0; i < N; i++){
                printf("\nEmployee : %d\n", i+1);
                read(&emp[i]);
        }
        sort(emp, N);
        /* displaying sorted employees */
        printf("\n SORTED EMPLOYEES:\n\n");
        for(i = 0; i < N; i++){
                display(emp[i]);
        }
        return 0;
}
void read(struct employee *e){
```

```
	printf("Enter name:");
	scanf(" %[^\n]", e->name);
	printf("Enter address:");
	scanf(" %[^\n]", e->address);
	printf("Enter salary:");
	scanf("%f", &e->salary);
}
void display(struct employee e){
	printf("\nName : %s\n",e.name);
	printf("Address : %s\n",e.address);
	printf("Salary : %f\n",e.salary);
}
int find_length(char *str){
	int i;
	for(i = 0; *(str + i) != '\0'; i++){
	}
	return i;
}
/* function that copares two strings */
int compare_string(char *str1, char *str2){
	int i, l1, l2;
	l1 = find_length(str1);
	l2 = find_length(str2);
	for(i = 0; *(str1+i) != '\0' || *(str2 + i) != '\0'; i++){
		if(*(str1+i) > *(str2+i)){
			return 1;
		}else if(*(str1+i) < *(str2+i)){
			return -1;
		}
	}
	if(l1 == l2){
		return 0;
	}else{
		return -1;
	}
}
void sort(struct employee e[], int size){
	struct employee temp;
```

```
    int i, j;
    for(i = 0; i < size - 1; i++){
        for(j = i + 1; j < size; j++){
            if(compare_string(e[i].name, e[j].name) == 1){
                temp = e[i];
                e[i] = e[j];
                e[j] = temp;
            }
        }
    }
}
```


10.6 Union

Unions are same like structures and follows the same syntax, and only the difference is in term of storage. In structure the members has its own storage location whereas all the members of the union uses the same location. Although, union can have many members of different types,, it can handle only one member at a time. The size of the structure is the sum of size of all the members of its elements, whereas the size of the union is the size of the largest member of the union. Union can be defined as:

```
union union_name
{
        data_typte1 member_1;
        data_typte2 member_2;
        ..............................
        ..............................
};
```

And can be declared as:

```
union union_name var_name;
```

10.7 structure vs union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Reference: https://www.geeksforgeeks.org/difference-structure-union-c/

Chapter-11
Data Files/ File Handling

11.1 Introduction

Meanwhile, the data was written to standard output device (monitor) and read from the standard input device (keyboard). The mentioned mechanism is convenient for small programs that uses very few lines of codes and little amount of memory. However, with large amount of data, the information has to be written to or read from auxiliary device. Such information is stored on the device in the form of a data file.
Such a mechanism gives us a platform to store the large amount of data permanently and in suitable format, so that we can retrieve any of the information from large pool of data effortlessly. C provides high level functions for such operations.

11.2 Types of files

There are two kinds of file in which data can be stored in two ways either in characters coded in their ASCII character or in binary format. They are

1. Text files
   A text file contains only the text information like alphabets, digits and special symbols. The ASCII codes of these characters are stored in these files. It only uses 7 bits allowing 8 bit to be zero.
2. Binary Files
   A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

The data contained in the text files can be recognized by the word processor while the binary files can not.

11.3 File Operations

11.3.1 naming a file

While working with files, first we need to name the file or identify the name of file. The path for the name of the file is optional. If we mention the path it will use the same path as specified and if not mentioned then the relative path will be used.

11.3.2 opening a file

In C, the another step to work with file is opening a file. Data Structure called **FILE** is already defined in C. Therefore, all files should be declared as type FILE before they are used. **FILE** is defined data type. For the operation of the file, we have to declare the FILE pointer and then the file can be opened using function **fopen().** Its declaration is:

FILE *fopen(const char *filename, const char *mode);

fopen returns FILE pointer on successful opening.
The general syntax is:

FILE *file_pointer_name;
file_pointer_name = fopen(“file_name”, “mode”); /* later we will discuss the modes*/
For example: If we want to open file called “file.txt” in read mode then it looks like.
FILE *fptr;
fptr = fopen(“file.txt”, “r”);

11.3.3 Modes

Mode	Meaning	Description
r	Open for reading	The file must exist. if the file doesn't exist fopen() returns **NULL**, And the cursor is located at the beginning.
rb	Opening for reading in binary mode.	The file must exist. if the file doesn't exist fopen() returns **NULL**, And the cursor is located at the beginning.
w	Open for writing	If the file exists, the contents are overwritten. And the cursor is located at the beginning. If the file doesn't exist, the new file will be created.
wb	Open for writing in binary mode	If the file exists, the contents are overwritten. And the cursor is located at the beginning. If the file doesn't exist, the new file will be created.
a	Open for appending, data is added to end of file.	If the file doesn't exist, new file will be created. And the cursor is located at the end if the file exists.
ab	Open for appending in binary mode.	If the file doesn't exist, new file will be created. And the cursor is located at the end if the file exists.
r+	Open for both reading and writing.	If the file doesn't exist, then returns NULL. The cursor is located at the beginning.
rb+ or r+b	Open for both reading and writing in binary mode.	The file must exist. If the file doesn't exist, then returns NULL. The cursor is located at the beginning.
w+	Opens for reading and writing.	If the file exists then the contents are overwritten, otherwise new empty file is created. The cursor is located at the beginning.
wb+ or w+b	Opens for reading and writing in binary mode.	If the file exists then the contents are overwritten, otherwise new empty file is created. The cursor is located at the beginning.
a+	Open for both reading and writing.	If the file exists the data is appended, the cursor is located at the end. If the file doesn't exist then new file is created.
ab+ or a+b	Open for both reading and writing in binary mode.	If the file exists the data is appended, the cursor is located at the end. If the file doesn't exist then new file is created.

11.3.4 writing the data to a file

The followings are the function available for writing the data to a file.

11.3.4.1 putc()/fputc()

putc() and **fputc()** is used to write a single character at a time to a given file. It writes the given character at the position denoted by the file pointer and then file pointer advances. The function returns the character written in the case of successful write operation or else error EOF is returned.

int putc(int ch, FILE *fptr);
ch → character to be written.
fptr → pointer to a file object that identifies the stream where character is to be written.

11.3.4.2 fputs()

Write the specified string not including the null character. **fputs()** has the following declaration.
int fputs(const char *str, FILE *stream);

str → string that is to be written.
stream → pointer to the FILE object that identifies the stream where the string is to be written.
This function returns non-negative value on success full writing, or else on failure returns EOF.

11.3.4.3 putw()
This function is used to write the integer to a file. Its declaration is:
int putw(int number, FILE *fptr);
number → integer number to be written,
fptr → file pointer

11.3.4.4 fprintf()
Likewise the printf() function writes formatted output to the console. This function writes a formatted output to a file stream. Its declaration is:
int fprintf(FILE *fptr, const char *format, arg_list…….);
fptr → pointer to a file object that identifies the file stream.
format → format specifier, that is same like for printf() function

For example:

```
FILE * fp;

fp = fopen ("file.txt", "w+");

fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
```


11.3.5 reading the data from a file
The followings are the functions that are available for the reading from the file.

11.3.4.1 getc()/fgetc()
This function reads or gets the single character at a time from the file. It returns the character present at position indicated by file pointer. After reading the character, the file pointer is advanced to next character. If pointer is at end of file or if an error occurs EOF file is returned by this function.
Its prototype is:
int fgetc(FILE *fptr);
fptr → pointer to a file object that identifies the stream on which the operation is to be performed.

11.3.4.2 fgets()
It reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
The prototype is:
char *fgets(char *str, int n, FILE *stream)
str → pointer to an array of chars where the string read is copied.
n → maximum number of characters to be read to the str.
stream → pointer to an FILE object that identifies an input stream.
The function returns str.

11.3.4.1 getw()
Reads an integer value from a file. Its prototype is:
int getw(FILE *fptr);

11.3.4.1 fscanf()
Likewise, the function scanf() function reads the values from the console to the variable.This function reads the formatted input from a stream. Its prototype is:
int fscanf(FILE *stream, const char *format, variable_list……..);

stream → pointer to a file object, that identifies the file stream.
format → format specifiers

11.3.6 closing a file

After any of the operation to a file, it is better to close the file for safety. The function fclose() can be used to close the file. The general syntax is:
fclose(file_pointer);

11.4 Reading and writing to a binary file

The function fread() and fwrite() can be used to read and write to a binary file respectively.

11.4.1 Writing to a binary file

The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write. The prototype is:

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

ptr → pointer to the array of elements to be written
size →The size of each element to be written
nmemb →Number of elements
stream →pointer to file object that specifies an output stream.

This function returns the total number of elements successfully returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, it will show an error.

11.4.2 Reading from a binary file

The function reads the data from the given stream. It has the following prototype.
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

ptr → this is a pointer to a block of memory with a minimum size of size * nmemb byte.
size → this is the size of the each element to be read
nmemb → this is the number of element, each of size size.
stream → this is the pointer to a FILE object that specifies an input stream.

The total number of elements successfully read are returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

11.5 Random access in file

11.5.1 fseek() function

This function is used for seeking the pointer position in the file at the specified byte. It has the following declaration.
int fseek(FILE *stream, long int offset, int whence);
stream → this is the pointer to a FILE object that identifies the stream
offset → this is the number of bytes to offset from whence.
whence → this is the position from where offset is added. It can be specified by one of the following constant.

1. **SEEK_SET**
   Beginning of a file, or 0
2. **SEEK_CUR**
   Current position of a file, or 1
3. **SEEK_END**
   End of a file, or 2

Returns zero on successful, else returns non-zero.

11.5.2 ftell() function

This function returns the current file position of the given stream. Its prototype is:
long int ftell(FILE *stream);

If the error occurs this function returns -1L.

11.5.2 rewind() function
Sets the cursor position to the beginning of the file stream.
Its prototype is:
void rewind(FILE *stream);

11.6 Error handling during the file operations

11.6.1 feof()
Tests the end of the file of the given stream. The declaration for the function is
int feof(FILE *stream);

Returns non-zero if EOF associated with the file stream is set, else return zero.

11.6.1 ferror()
This function tests the error indicator for the given stream. The declaration is:
int ferror(FILE *stream);

If the error indicator associated with the stream was set, the function returns a non-zero value else, it returns a zero value.

Please refer the lab solutions for file handling (lab 10) from site rkamal.com.np

Chapter-12
Introduction to graphics

Graphics Functions
to be added

Graphics color codes

Color	Numeric Value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8

LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

Graphics Driver Constant

graphics_drivers constant	Numeric value
DETECT	0 (requests autodetect)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Please refer the lab solutions for Graphics Programming (lab 11) from site rkamal.com.np

References:

1. ***Programming in ANSI – E. Balagurusamy(seventh edition)***