Ch-1

Pre-Programming Techniques

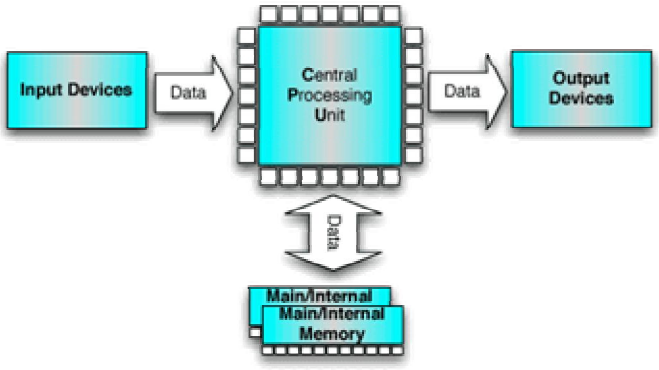
- ✓ Introduction
- ✓ Algorithm
- ✓ Flow Chart
- ✓ Dry Run
- ✓ Examples of Algorithm and Flow Chart

❖ Introduction to Computer & Programming

➢ Introduction

➔ A **computer** is a programmable machine designed to sequentially and automatically carry out a sequence of arithmetic or logical operations.

➢ Basic Block Diagram



➢ Concept of Hardware and Software

➔ Our computer system can be divided into 2 parts.

1. Hardware
2. Software

1.)Hardware

➔ All the physical parts of the computer are known as "Hardware".

➔ We can say that all the parts of the computer that have physical existence are known as hardware.

2.) Software

➔ All the Logical parts of the computer are known as "Software".

➔ We can say that all the parts of the computer that have logical existence are known as software.

➔ The software can be divided into 2 parts:

a. Application Software
b. System Software

➢ Concept of Basic types of Software

➔ Software is a set of instructions, programs which enable the computer to perform specified task.

➔ In other words, software is nothing but binary code instructions which control the hardware.

➔ The software can be divided into 2 parts:

1. Application Software
2. System Software

➔ Let's learn deeply:

1. Application Software

✓ Application software on the other hand, performs specific tasks for the computer user.

✓ Application software is a subclass of computer software that employs the capabilities of a computer directly and thoroughly to a task that the user wishes to perform.

✓ **Application software**, also known as an **application** or an "**app**", is computer software designed to help the user to perform singular or multiple related specific tasks.

✓ It helps to solve problems in the real world.

✓ Examples…

enterprise software
accounting software
office suites
graphics software and
media players

2. System Software

✓ System software is a set of programs that manage the resources of a computer system, so that they are used in an optimal fashion, provide routine services such as copying data from one file to another and assist in the development of applications programs.

✓ System software consists of programs that assist the computer in the efficient control, support, development and execution of application programs.

✓ The system software controls the execution of the application software and provides other support functions such as data storage.

✓ Examples…

Windows XP, 7, 2000, etc…
LINUX / UNIX
DOS

➢ Introduction to Programming & Programming Language

➔ The Computer language can be divided into 2 categories…

1) Low Level Language
2) High Level Language
3) Middle Level Language

1.) Low Level Language

- ✓ The low level language is made up of 0s and 1s and require less compile time but there are so many chances of errors while developing programs.

2.) High Level Language:

- ✓ The High-level languages are just like an English language and require more compile time but there are fewer chances of errors while developing programs.
- ✓ The High-level language Programs require the conversion programs.

1. Compiler

➔ It checks your whole program at a time and list out all the errors of your program.

2. Interpreter

➔ It checks one line of your program at a time and indicates error at that line.

➔ If you not solving the error of that line the interpreter can't move on next line.

3.) Middle level language

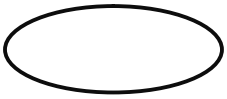
- ✓ The Middle level language is developed to maintain communication between Low and High Level Language.
- ✓ Example…
  - ✍ 'C' Language
  - ✍ C++

➢ Flow Chart , Algorithms and Dry Run

➲ Flow Chart

➔ Flow Chart is a diagrammatic representation to get solution of any problem.
➔ Flow Charts are drawn in earlier time to get solution.
➔ In terms of programming language, the Flow charts are used to represent any problems graphically.
➔ In short, the logic techniques of solving problems are known as "Flow Chart".
➔ Flow Chart facilitates communication between programmers and business people.
➔ Once the flow chart is drawn, it becomes easy to write the program in High Level Language.
➔ There are several symbols, which are used to draw flowcharts:

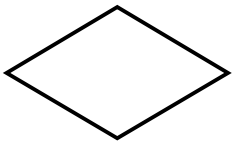
1. Terminator/Oval:
✓ This symbol is used to represent the starting and ending point of "Flow Chart".
✓ This symbol may contain the words like Start, Begin, Stop, and End Etc.



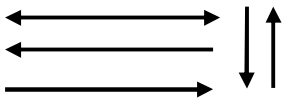
2. Rectangle:
✓ This symbol is used to represent Process associated with the problem.
✓ This symbol is also used to declare the variables.

3. Input/Output:
✓ This symbol is used to represent All Inputs and Outputs, which are associated with the flowcharts.

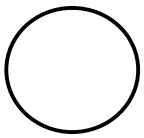
4. Diamond:
✓ This symbol is used to represent the conditions associated with the flowcharts.
✓ This symbol always contains the conditions or expressions.



5. Flow Lines:
✓ The flow lines are used to indicate the control flow of the flow chart.
✓ Generally the flow of control may be top to bottom or left to right.



6. Connector:
✓ This symbol is used to connect the multiway flow of the control, and direct the control to the particular point.

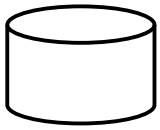


7. Magnetic Tape (Sequential Access Storage)

- ✓ Although it looks like a 'Q', the symbol is supposed to look like a reel of tape.

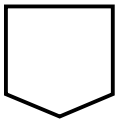
8. Magnetic Disk (Database)

- ✓ The most universally recognizable symbol for a data storage location, this flowchart shape depicts a database.



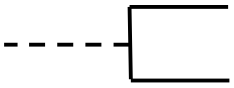
9. Off-page Connector

- ✓ It is used to signify a connection to a process held on another sheet screen.



10. Annotation

- ✓ It is used to keep data secret from outside world.



11. Display

- ✓ Indicates a process step where information is displayed to a person (e.g., PC user, machine operator).

➲ Algorithm

➔ Algorithms are developed on the basis of Flowcharts.
➔ To make a computer do anything, you have to write a computer program.
➔ To write a computer program, you have to tell the computer, step by step, exactly what you want to do it.
➔ Algorithms are just the textual presentation of the flowchart.
➔ The algorithm is the basic technique used to get the job done.
➔ In short, the step by step representation of any problem is known as algorithms.
➔ Types of Algorithm:

1.) The taxi algorithm
1. Go to the taxi stand.
2. Get in a taxi.
3. Give the driver my address.

2.) The call-me algorithm
1. When your plane arrives, call my cell phone.
2. Meet me outside baggage claim.

3.) The rent-a-car algorithm

1. Take the shuttle to the rental car place.
2. Rent a car.
3. Follow the directions to get to my house.

4.) The bus algorithm

1. Outside baggage claim, catch bus number 70.
2. Transfer to bus 14 on Main Street.
3. Get off Elm Street.
4. Walk two blocks north to my house.

➔ All four these algorithms accomplish exactly the same goal, but each algorithm does it in completely different travel time.

➔ You should choose the algorithm based on the circumstances.

➔ Each algorithm has advantages and disadvantages in different situations.

➔ Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists.

➔ There are mainly 5 different algorithms are used for algorithms:

1. Bin sort
2. Bubble sort
3. Merge sort
4. Quick sort
5. Shell sort

➔ By knowing the strengths and weaknesses of the different algorithms, you pick the best one for the task at hand.

➔ In short, We can say that…

Flow Chart = Graphics

Algorithm= Text

➲ Dry Run

➔ A dry run is a testing process where the effects of a possible failure are intentionally mitigated.

➔ In computer programming, a dry run is a mental run of a computer program, where the computer programmer examines the source code one step at a time and determines what it will do when run.

➔ In theatrical computer science, a dry run is a mental run of an algorithm, sometimes explained in pseudocode, where the computer scientist examines the algorithm's procedures one step at a time.

➔ In both uses, the dry run is frequently assisted by a table with the program or algoritm's variables on the top.

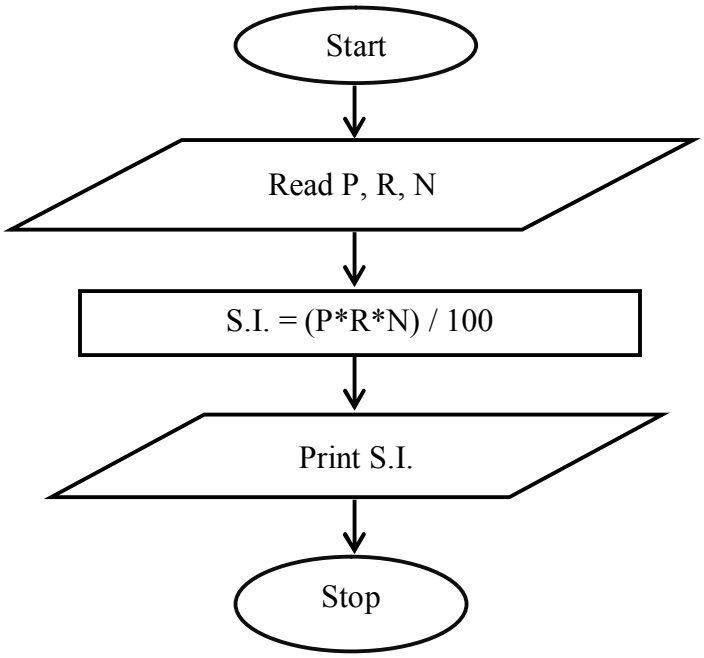
➔ The useage of DRY RUN in accetance procedures is meant following:

The firm(which is a subcontractor) must perform a complete test of the system it has to deliver before the actual acceptance from the contractor side.

➲ Example

1.) Draw a FLOW CHART, create an ALGORITHM and perform DRY RUN to find SIMPLE INTEREST.

Flow Chart:



Algorithm:

Step 1: Begin

Step 2: Initialize P, R, N

Step 3: Input the value of P

Step 4: Input the value of R

Step 5: Input the value of N

Step 6: Calculate the SIMPLE INTEREST by using (P*R*N)/100 formula.

Step 7: Print the value of SIMPLE INTEREST

Step 8: End

Dry Run:

➔ L1 declares 3 variables, Principal Amount, Rate, No. of years.
➔ L2 initialises all three variables with garbage value.
➔ L3 ask user to enter PRINCIPAL AMOUNT.
➔ L4 assign user input to P variable.

After execution	P	R	N	Result
Line2	xxx	xxx	xxx	
Line 4	10000	xxx	xxx	
Line 6	10000	2.5	xxx	
Line 8	10000	2.5	5	
Line 9	SI=10000*2.5*5/100			

➔ L5 ask user to enter RATE (per year).
➔ L6 assign user input to R variable.
➔ L7 ask user to enter NO. OF YEARS.
➔ L8 assign user input to N variable.
➔ L9 performs an action to get Simple Interest.
➔ L10 print Simple Interest.

Line 10	10000	2.5	5	1250

Ch-2

History & Overview of C Language

- ✓ History of C Language
- ✓ Features of C Language
- ✓ C Program Structure
- ✓ Character Set
- ✓ C Tokens
- ✓ Hierarchy of Operators
- ✓ Type Casting
- ✓ Data Types

❖ History of C Language

➔ C Language has emerged as the most used programming language for software devlopers.
➔ The language gets name from being the successor to the B language (an interpreter based language).
➔ It was designed by DENNIS RITCHIE in 1972 as the system language for UNIX operating system on a PDP-II computer at AT & T Bell laboratories.

➔ At the time of 1960's the need of programming language was raised for almost specific purpose.
➔ COBOL was used for commercial purpose and FORTRAN was used for Engineering and Scientific Purpose etc…
➔ There are some limitations in these types of languages.

➔ To reduce these limitations, a new language called ALGOL was developed and again to reduce the limitations of ALGOL the language called Combine Programming Language (CPL) was developed by Cambridge University.
➔ However CPL was hard to learn and difficult to implement, so it was also turned out.
➔ Martin Richards at Cambridge University developed Basic Combined Programming Language (BCPL) by adding good features in the CPL.
➔ But unfortunately this language was also turned out because of less powerful.
➔ At the same time **Ken Thompson** was developed the language called **B language** at AT & T's Bell labs.

➔ Dennis Ritchie inherited the features of language called B and BCPL and added some more features of his own and developed a new language called **C.**
➔ So, In Nutshell……
- ꕥ C is the Middle level programming language.
- ꕥ Developed By Dennis Ritchie.
- ꕥ In the year 1972.
- ꕥ At the place called AT & T's Bell labs, in USA.

➔ In beginning C language was not accepted by all programmers and so it was not so much popular.
➔ Brain Keningham and Dennis Ritchie both made some additions and changes in it and then published a very successful book named "**The C Programming Language**".

➔ ANSI committee has made some standards for it and made chages in C, so it can run over variety of operating systems like UNIX, DOS, MAC etc… and then approved C as ANSI-C in 1989, lately ISO committee also approved it in 1990.

➔ Note:

- ꕥ C have both a relatively good programming efficiency and relatively good machine efficiency.
- ꕥ That's why it is termed as "**Middle Level Language**".

➔ Remember it…

1960	ALGOL	International Group
1967	BCPL	Martin Richards
1970	B Language	Ken Thompson
1972	Traditional C	Dennis Richie
1978	K & R C	Berian Kerningham & Dennis Ritchie
1989	ANSI C	ANSI committee
1990	ANSI/ISO C	ISO committee

❖ Features of C Language

➔ 'C' is the Programming language.
➔ It is easy to understand.
➔ It is easy to develop logic.
➔ It is a sequential programming language.
➔ There are several reasons why many computer professionals feel that C is at the top of the list:

➲ FLEXIBILITY

➔ C is a powerful and flexible language.
➔ C is used for projects as diverse as operating system, word processor, graphics, spreadsheets and even compilers for other language.
➔ C is a popular language preferred by professional programmers.
➔ As a result, a wide variety of C compilers and helpful accessories are available.

➲ PORTABILITY

➔ C is a portable language.
➔ Portable means that a C program written for one computer system can be run on another system with little or no modification.
➔ Portability is enhanced by the ANSI standard by C, the set of rules for C compilers.

➲ **COMPACTNESS**

- ➔ C is a language of few words, containing only a handful terms, called keywords, which serve as the base on which the language's functionality is built.
- ➔ You might think that a language with more keyword would be more powerful.
- ➔ This isn't true. As you will find that it can be programmed to do any task.

➲ **REUSABILITY**

- ➔ C is modular, C code can and should be written in routine called functions.
- ➔ These functions can be reused in other applications or programs.
- ➔ By passing pieces of information to the function, you can create useful, reusable code.

❖ C Program Structure

- ➔ C program can be viewed as group of building blocks called functions.
- ➔ A function is a subroutine that may include one or more statement designed to perform a specific task.
- ➔ A C program may contain one or more section as shown here:

```
Documentation Section
Link Section
Definition Section
Global declaration Section
Main() function section
{
    Declaration Part
    Executable Part
}
Subprogram Section
        Function1 ()
        {
        }
        Function2 ()
        {
        }
```

Documentation Section

This section contains set of comments lines consist of details like program name, author name and purpose or functionality of the program.

Link Section

This section consists of instructions to be given to the compiler to link functions from the system library.
For example if you want to use some mathematical function then you have to define link for math.h file in link section.
For Example

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

Definition Section

This section defines all the symbolic constants. For example PI=3.14. By defining symbolic constant one can use these symbolic constant instead of constant value.

```
# define PI 3.14
# define Temp 35
```

Global Declaration Section

This section contains the declaration of variables which are used by more than one function of the program.

Main Function Section

A main() function is a heart of any 'C' language program. Any C program is made up of 1 or more then 1 function. If there is only 1 function in the program then it must be the main program. An execution of any C program starts with main() function.

Subprogram Or Sub Function Section

They are the code section which is defined outside the boundary of main function. This function can be called from any point or anywhere from the program. Generally they are define to solve some frequent tasks.

Note: Section except the main() section may be absent when they are not required.

❖ Character Set

➔ C language allows many characters and symbols.
➔ The characters in C language is divided into 4 parts.
➔ The character set in C language is as follows:

CHARACTER SET	SYMBOLS
Letters	Upper Case → A to Z Lower Case → a to z
Digits	All decimals → 0 to 9
Special Characters	, Comma . Dot ; Semicolon : Colon ? Question Mark ' Single Quote "Double Quote ! Exclamation | Pipe / Slash \ Back Slash ~ Tiled _ Under Score $ Dollar % Percentage & Ampersand ^ Caret * Asterisk - Minus + Plus < Opening Angular Bracket > Closing Angular Bracket (Left Parenthesis) Right Parenthesis [Left Bracket] Right Bracket { Left Brace } Right Brace # Number Sign (Hash)
White spaces	Blank Space Horizontal Tab Carriage Return New Line Form Feed

❖ C Tokens

➔ Individual words and punctuation marks are called TOKENS.
➔ Similarly, in a 'C' program the smallest individual units are known as 'C' tokens.
➔ 'C' has 6 types of tokens:
1.) K – Keywords
2.) I – Identifiers
3.) S – Special Symbols
4.) C – Constants
5.) O – Operators
6.) S – Strings

➲ Keywords

➔ The keywords are the words whose meaning is already explained to the c compiler.
➔ The key words are also known as RESERVED WORDS.
➔ There are total 32 keywords available in C.
➔ They are as given here:

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

➲ Identifiers

➔ Identifiers refer to the names of VARIABLES, FUNCTIONS, and ARRAYS.
➔ These are user-defined names and consist of a sequence of letters, underscore (_) and digits, with a letter as a first character.

⇒ VARIABLES

➔ A variable is the name given to the memory location.
➔ During the execution of program the values of variables may be changed/altered.
➔ We all know that we can't access the RAM locations directly,
Now what we can do for storing our data?
➔ So, C supports the concepts of Variable…
➔ There are some rules which must be following while defining variable:

1. The variable name must begin with alphabets.
2. The variable name should be meaningful. That means the variable should clearly indicate the purpose.
3. No special symbols rather than underscore (_) is allowed.
4. The variable name must not be a keyword.
5. No commas or blanks are allowed while defining a variable.
6. The variable name may be combination of alphabets, digits and underscore.

➲ Special Symbols

➔ There are some symbols which are used to define array, function, and etc…

➔ These special symbols are as below:

1.) ( ) - Function Call
2.) [ ] - Array
3.) { } - Definition of Function
4.) < > - Link to Header File

➲ Constants

➔ The Values, which are stored inside the variables, are known as Constants.

➔ During the execution of program the constant remains fixed.

➔ C Supports various types of constant, which are as given below:

1.) Numeric constant
   i. Integer Constant
   ii. Real Constant
2.) Character Constant
   i. Single Character Constant
   ii. String Constant

1) Numeric Constant

➔ As its name suggests the numeric constants are made up of digits.

➔ There are 2 types of numeric constants as given below:

i. Integer Constant

➔ There are some rules for defining integer constants.

➔ Rules:

1. An Integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative.
4. If no sign proceeds, an integer constants assumed is positive.
5. No commas or blanks are allowed within an integer constant.
6. Example: 436,45 10

ii. Real Constant

➔ There are some rules for defining a real constant:

1. A Real constant must have at least on digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. If no sign proceeds, an integer constants assumed is positive.
5. No commas or blanks are allowed within an integer constant.
6. Example: 134.98, 235.67

2) Character Constant

➔ As its name suggests the character constant always contains one character or no. of characters.

➔ There are 2 types of character constants.

i. Single Character Constant

➔ There are some rules for defining single character constants.

➔ They are as given below.

➔ Rules:

1) A character constant is either a single alpha bates a single digit or a single special symbol enclosed within single inverted commas.
2) Example: □ 'A', ' ' , '*'

ii. String Constant

➔ The rules of defining string constants are as given below.

➔ Rules:

2) The string constants are always enclosed within double inverted commas.
3) The string constants are just combination of characters, digits, and special symbols.
4) Example: - "Welcome", "Hello"

➲ Operators

➔ Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

➔ Operators are just the symbol that can perform manipulations on the data.

➔ C has rich set of operators.

➔ There are several types of operators, they are as given below:

1) Arithmetic Operators

➔ Arithmetic operators are used to perform arithmetic operations on data.

➔ The Arithmetic operators are as given below:

Operator	Meaning
+	Addition OR Unary Plus
-	Subtraction OR Unary Minus
*	Multiplication
/	Division
%	Modulo Division

➔ Example: months = days / 30;

2) Assignment Operators

➔ Assignment operators are used to assign a value to the variables.

➔ "=" Sign is used to perform an assignment operation.

➔ Arithmetic assignment operators are also used in combination to simplify the use of both operators.

➔ The combinations of both operators are known as "Arithmetic Assignment Operators" or "Short Hand Operators".

➔ 'C' has a set of 'shorthand' assignment operators of the form

variable operator= expression;

Statement with simple Assignment Operator	Statement with Shorthand Operator
a = a + 1	a += 1
a = a – 1	a -= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

3) Logical Operators

➔ Logical operators are used to combine to or more expressions.

➔ In order to combine two or more the one relational operator, the logical operators are used.

➔ There are 3 logical operators as given below:

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

➔ For example: a > b && x == 100

➔ An expression of this kind, which combines two or more relational expressions, is termed as a LOGICAL EXPRESSION or a COMPOUND RELATIONAL EXPRESSION.

➔ Like, Simple relation expressions, a LOGICAL EXPRESSION also yield, a value of 1 or 0, according to the TRUTH TABLE.

Op-1	Op-2	Value of the expression	
		Op-1 && Op-2	Op-1 || Op-2
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

4) Unary Operators

➔ The unary operators can operand one operand only at a time.

➔ They are just used to increase or decrease the value of operand.

➔ They are always used with integer variable but cannot be used with float variables.

➔ They are listed below:

- ➲ Increment (++)
- ➲ Decrement (--)

➔ They are increase / decrease the value of variable by 1,which is by default.

➔ Example: ++m or m++

(++m is equivalent to m = m+1 OR m+= 1)

5) Comparison Operators OR Relational Operators

➔ The comparison operators are also known as "Relational operators".

➔ Because they are used to show the relationship between 2 variables and variables and constants. They evaluate to either TRUE or FALSE.

➔ Given below are the comparison operators:

Operator	Meaning
<	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

6) Conditional Operator

➔ A ternary expression operator pair " ?: " is available to construct conditional expressions.

➔ Syntax

Exp1 ? Exp2 : Exp3;

where,, exp1, exp2, exp3 are expressions.

➔ The operator ?: works as follows:

- ➲ Exp1 is evaluated first.
- ➲ If it is NON-ZERO (TRUE), then the expression Exp2 is evaluated and becomes the value of expression.
- ➲ If Exp1 is FALSE, Exp3 is evaluated and its value becomes the value of the expression.
- ➲ Note that only one of the expressions (either Exp2 or Exp3) is evaluated.

➔ It replaces simple if…else loop.

➔ Example…

```
if (a > b)
    x = a;          is replaced by      x = (a > b) ? a : b;
else
    x = b;
```

7) Bitwise Operators

➔ Some applications require the manipulation of individual bits within a word of memory.
➔ Assembly language or machine language is normally required for operations of this type.
➔ However C contains several special operators that allow such bitwise operations to be carried out easily and efficient.
➔ These bitwise operators can be divided into three general categories:
   1. The one's complement operator.
   2. The logical bitwise operators.
   3. The shift operators.
➔ It also contains several operators that combine bitwise operations with ordinary assignment.

➢ The One's Complement Operator

➔ The one's complement operator (~) is a unary operator that causes the bits of its operand to be inverted (i.e. reversed) so that is become so a so become is.
➔ This operator always precedes its operand. The operand must be an integer type quantity (including integer, long, short, unsigned, or char).
➔ Linearly the operand will be an unsigned octal or an unsigned hexadecimal quantity, though this is not a firm requirement.

Suppose int a=16 with 2 byte=16 bit pattern

it will be → 00000000 00010000

b=~a

the b will have 16 bit pattern like → 11111111 11101111

➢ Logical Bitwise Operators

➔ There are three logical bitwise operators:
   1. Bitwise and ($)
   2. Bitwise exclusive-or(^)
   3. Bitwise or (!)
➔ Each of these operators requires two integer type operands.
➔ The operations are carried out independently on each pair of corresponding bits within the two operands will be.
➔ Thus the least significant bits(i.e. the rightmost bits) within the two operands will be compared then the least significant bits. And so on.
➔ Until these comparisons are

B1	B2	B1&B2	B1|B2	B1 ^ b2
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1

0	0	0	0	0

➔ The associatively for each bitwise operator is left-to-right.

➢ Shift Operators

➔ The two bitwise shift operator is shift left (<<) and shift right(>>).
➔ Each operator requires two operands. The first is an integer-type operand that represents the bit pattern to be shifted.
➔ The second is an unsigned integer that indicates the number of displacements (i.e. whether the bits in the first operand will be shifted by 1 bit position, 2 bit position, 3 bit positions and so on).
➔ This value cannot exceed the number of bits associated with the word size of the first operand.
➔ The left-shift operator causes the left by the bits in the first operand to be shifted to the left by the number of position indicated by the second operand.
➔ The leftmost bit positions that become vacant will be tilled with us.

A = 01101101 10110111

A<<6 = 0110 1100 0000 = 0x5dc0

➔ All the bits originally assigned to are shifted to the left six places, as the arrows indicate.
➔ The leftmost 6 bits (originally 011011) are lost.
➔ The rightmost 6 bits positions are filled with 00 0000.
➔ The right shift operator causes all the bits in the first operand to be shifted to the right by the number of positions indicated by the second operand.
➔ The rightmost bits(i.e. the underflow bits) in the original bit pattern will be lost.
➔ If the bit pattern being shifted represents an unsigned integer, then the leftmost bit positions that become vacant will be filled with 0s.
➔ Hence, the behavior of the right-shift.
➔ When the first operand is an unsigned integer.

A= 0110 1101 1011 0111

A>>6 = 000 0001 1011 0110 = 061bc

➔ We see that all the bits originally assigned to are shifted to the right six places, as the arrows indicate.
➔ The rightmost 6 bits (originally 11 011) are lost. The leftmost 6 bits positions are filled with 00 0000.

8) Special Operators

➔ There are several operators which are used to perform some special task.
➔ Some are listed below:

1. **sizeof Operator**

2. comma (,) Operator

1. sizeof Operator

➔ The size of a compile time operator and when used with an operand, it returns the number of
bytes the operand occupies.

➔ The operand may be variable, a constant or a data type qualifies.

- **M = sizeof(sum);**
- **N= sizeof(long int);**
- **K= sizeof(253L);**

➔ The size of operator is normally used to determine the length of array and structures, when their sizes are not known to the programmer.

➔ It is used to allocate memory space dynamically to variable during execution of a program.

2. Comma(,) Operator

➔ The comma operators are use primarily in conjunction with for statement.

➔ This operator permits two different expressions to upper in situation where only one expression would ordinarily be used.

➔ For example, it is possible to write

For(expression la, expression lb; expression2; expression3) Statement

➔ Where expression la and lb are the two expressions, separated by the comma operator, where only one expression would normally appear.

➔ These two expression would typically initialize two separate indices that would be used simultaneously within the for loop.

➔ Similarly, a for statement might make use of the comma operator in the following manner.

For(expression 1a; expression2;expression3a, expression3b) statement;

➔ Here expression 3a and expression 3b separated by the comma operator, appear in place of the usual single expression.

➔ In this application the two separate expressions would typically be used to alter the two different indices used simultaneously within the loop.

➔ For example, one index might count forward while the order counts backward. In for loops.

for(n=1,m=10; n<=m; m++, n++)

❖ Hierarchy of Operators

➔ An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators.

High Priority ➔ *, /, %

Low Priority ➔ +, -

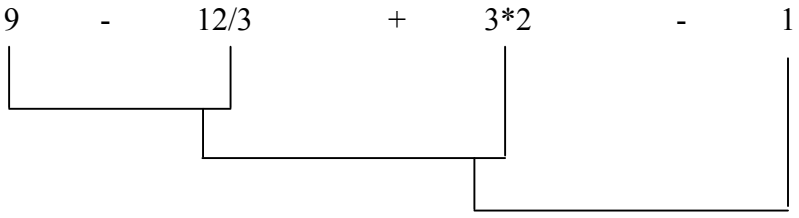
Ex.

x=9
y=12
z=3 then
ans=x-y/3+3*2-1

Here,,
ans=x-y/3+3*2-1
ans=9-12/3+3*2-1

Step 1: ans=9-4 + 3*2-1 1st pass
Step 2: ans=9-4+6-1
Step 3: ans=5+6-1 2nd pass
Step 4: ans=11-1
Step 5: ans=10

It works like this:



➔ Rules for evaluation of expression are as follows:

1. First paranthesized sub expression is evaluated from L to R. (bracket statements)
2. If parenthesis are nested, the evaluation begins with the inner most sub-expression.
3. Arithmetic expressions are evaluated from L to R using rules of PRECEDENCE.
4. When parenthesis are used, the expression within parenthesis assumes HIGHEST PRIORITY.

➢ Precedence and Associativity Table

➔ Precedence is a priority of the operator for evaluation or execution when there are more then 1 operator in a single expression or statement.

➔ If there are operators of same precedence then the expression will evaluated either from LEFT-TO-RIGHT or RIGHT-TO-LEFT., This is called an associativity of an operator.

Description	Operator	Associativity	Precedence
Function expression	( )	L TO R	1
Array expression	[ ]	L TO R	
Structure operator	->	L TO R	
Structure operator	.	R TO L	
Unary minus	-	R TO L	2

Increment/Decrement	++ - -	R TO L	
One's complement	~	R TO L	
Negation	!	R TO L	
Address of	&	R TO L	
Value of address	*	R TO L	
Type cast	(type)	R TO L	
Size in bytes	sizeof	R TO L	
Multiplication	*	L TO R	3
Division	/	L TO R	
Modulus	%	L TO R	
Addition	+	L TO R	4
Subtraction	-	L TO R	
Left Shift	<<	L TO R	5
Right Shift	>>	L TO R	
Less than	<	L TO R	6
Less than equal to	<=	L TO R	
Greater than	>	L TO R	
Greater than or equal to	>=	L TO R	
Equal to	= =	L TO R	7
Not equal to	!=	L TO R	
Bitwise AND	&	L TO R	8
Bitwise exclusive OR	^	L TO R	9
Bitwise inclusive OR	|	L TO R	10
Logical AND	&&	L TO R	11
Logical OR	||	L TO R	12
Conditional	? :	R TO L	13
Assignment	= ,*= ,/=, %= +=,-=,&=,|=,^=,!= <<=,>>=	R TO L	14
Comma	,	R TO L	15

❖ Type Casting

➔ In computer science, type conversion / type casting refers to changing an entity of one data type to another.

➔ There are two types of conversion;

1. Implicit Conversion
2. Explicit Casting

1.)Implicit Conversion

- ✓ It works in a way that a variable of data type that is smaller in length, transforming internally to variable of data type with no longer number length.
- ✓ It follows below hierarchy:

➲ short int → int → unsigned int → long int → unsigned long int → float → double → long double.

2.)Explicit Casting

- ✓ It has higher priority then automatic transformation.
- ✓ General declaration of explicit cast:

Syntax:

(data_type) operand

Operand can be variable or phrase.

Example:

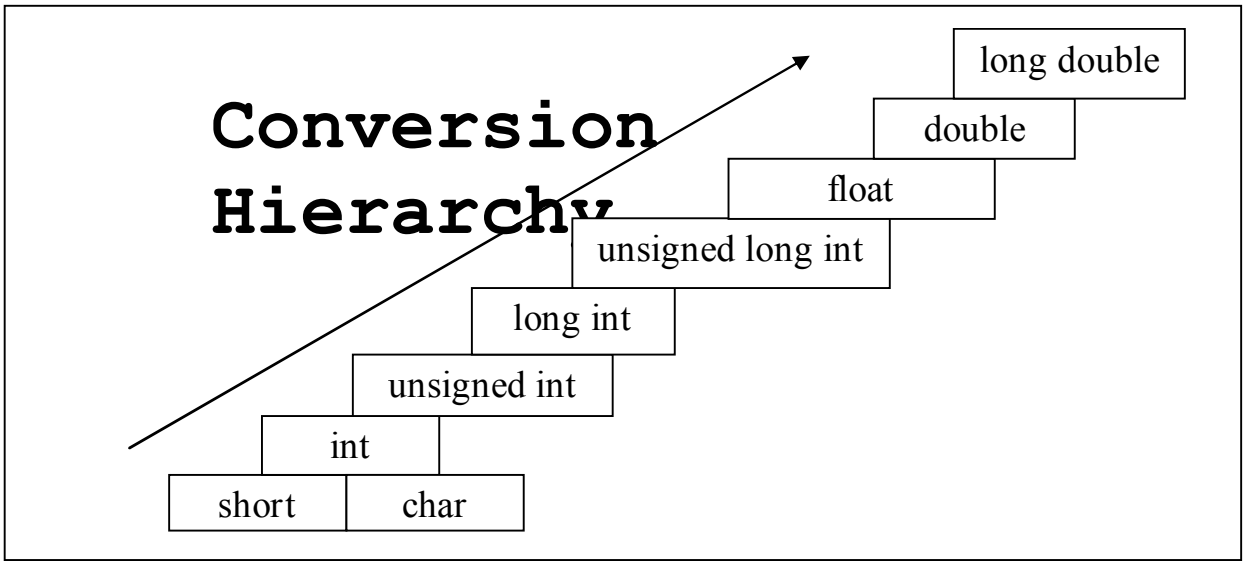
```
a = (int) c ;
b = (double) d + c ;
```

❖ Data Types

- ➔ The predefined identification of any variable is known as "Data type".
- ➔ The data type of any variable tells that what kind of values will be stored inside the variable.
- ➔ C has rich set of data types.
- ➔ Basically there are 4 data types, but all data types has some modifiers which helpful to extend the functionality of data types.
- ➔ The data types supported by C are as given below:

Data Type	Keyword equivalent	Size (In Bytes)	% Format	Range
Character	char	1 byte (8 bits)	%c	-128 to +127
Unsigned Character	unsigned char		%c	0 to 255
Signed Short Integer	signed short int OR short int OR short		%d	-128 to +127
Unsigned Short Integer	unsigned short int OR unsigned short		%d	0 to 255
Signed Integer	int OR signed int	2 bytes (16 bits)	%d	-32,768 to +32,767
Unsigned Integer	unsigned int OR unsigned		%u	0 to 65,535
Signed Long Integer	signed long int OR long int OR	4 bytes (32 bits)	%ld	-2,147,483,648 to +2,147,483,647

	long			
Unsigned Long Integer	unsigned long OR unsigned long		%lu	0 to 4,294,967,295
Floating point	float		%f	3.4E – 38 to 3.4E + 38
Double precision floating point	double	8 bytes (64 bits)	%lf	1.7E – 308 to 1.7E + 308
Extended double precision floating point	long double	10 bytes (80 bits)	%Lf	3.4E – 4932 to 1.1E + 4932



Ch-3

Various Control Structure

- ✓ Introduction
- ✓ Sequential Control Structure
- ✓ Selective Control Structure
- ✓ Iterative Control Structure
- ✓ Break Statement
- ✓ Continue Statement
- ✓ The goto Statement

❖ INTRODUCTION

➔ The term FLOW CONTROL refers to the order in which a programmer's statements are executed.

1. Sequential
2. Selection
3. Iterative

➔ The normal flow of control for all program is sequential.

➔ The SELECTION statements allow the programmer to alter the normal sequential flow of control.

➔ The ITERATIVE statements provide us ability to go back and repeat a set of statements.

❖ SEQUENTIAL CONTROL STRUCTURE

➔ The Sequential Control Structure is nothing but a linear structure so we can call sequential programming as the linear proframming.

➔ Non-modular in nature.

➔ Reusabiltiy of code is not possible.

➔ Difficult to understand.

/* Write A Program to print Simple Interest. */

```
#include<stdio.h>
#include<conio.h>
void main()
{
        float P,R,N,SI;
        clrscr();
        printf("Enter Principal amount: ");
        scanf("%f",&P);
        printf("Enter Principal amount: ");
        scanf("%f",&R);
        printf("Enter Principal amount: ");
        scanf("%f",&N);

        SI=(P*R*N) / 100 ;
        printf("\n\t  Simple Interest: %.3f",SI);
        getch();
}
```

❖ SELECTIVE CONTROL STRUCTURE

➔ To perform such things C supports decision making statements that are as given below:

A. if statement
B. switch statement
C. Conditional Operator statement

➔ These statements are purely known as "Decision Making".

➔ They are also 'control' the flow of execution, they are also known as "Control Statement".

A.) if statement

➲ DECISION MAKING USING 'IF' STATEMENT

➔ The if statement is the powerful decision making statement and is used to control the flow of execution of the statements.

➔ C uses the keyword if to implement the decision control instruction.

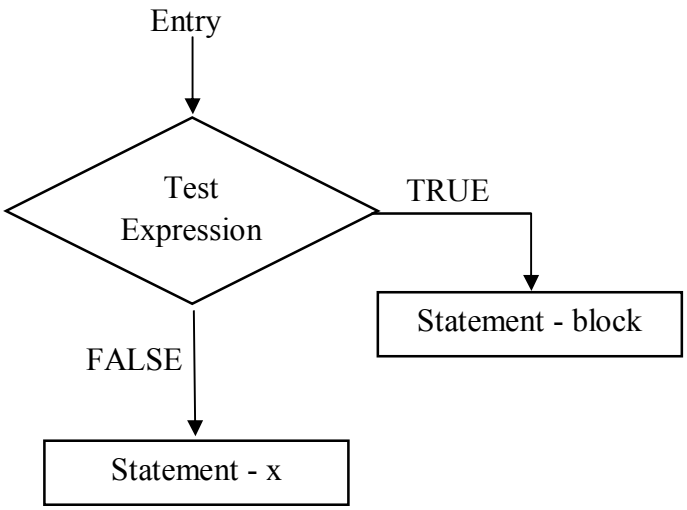
➔ It is basically a 2 way decision making statement.

➔ Syntax:

```
if(test expression)
{
        statement block;
}
```

➔ Example:

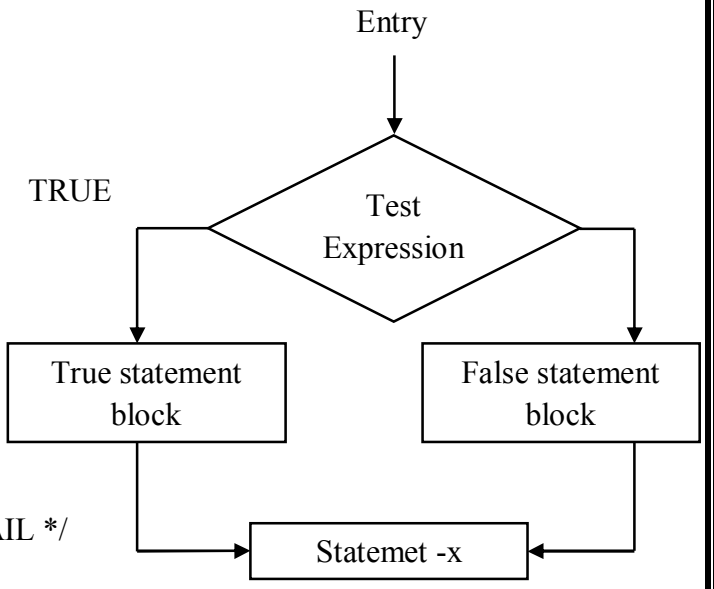
```
/* Program to find check the person is MALE or FEMALE. */
#include<stdio.h>
#include<conio.h>
void main()
{
    char code;
    clrscr();
    printf("Enter your gender: ");
    scanf("%c",&code);
    if(code == 'M' || code == 'm')
        printf("The person is MALE.");
    if(code == 'F' || code == 'f' )
        printf("The person is FEMALE.");

    getch();
}
```



➲ DECISION MAKING USING IF…ELSE STATEMENT

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

➔ The general form of if…else is as given below:



➔ Syntax:

```
if(test expression)
{
        true-block statements;
}
else
{
        false-block statements;
}
Statement-x;
```

➔ Example:

```
/* WAP to check whether you are PASS or FAIL */
#include<stdio.h>
#include<conio.h>
void main()
{
        float per;
        clrscr();
        printf("Enter Your Percentage: ");
        scanf("%f",&per);
        if(per>=40.00)
                printf("You are PASS");
        else
                printf("You are FAIL");
        printf("You have scored: %.3f",per);
        getch();
}
```

➲ DECISION MAKING WITH NESTED IF… ELSE STATEMENT

➔ When a one if..else statement is within the another if..else is known as "Nesting of If..Else".

➔ The general form of the nested if..else is as given below.

➔ Syntax:

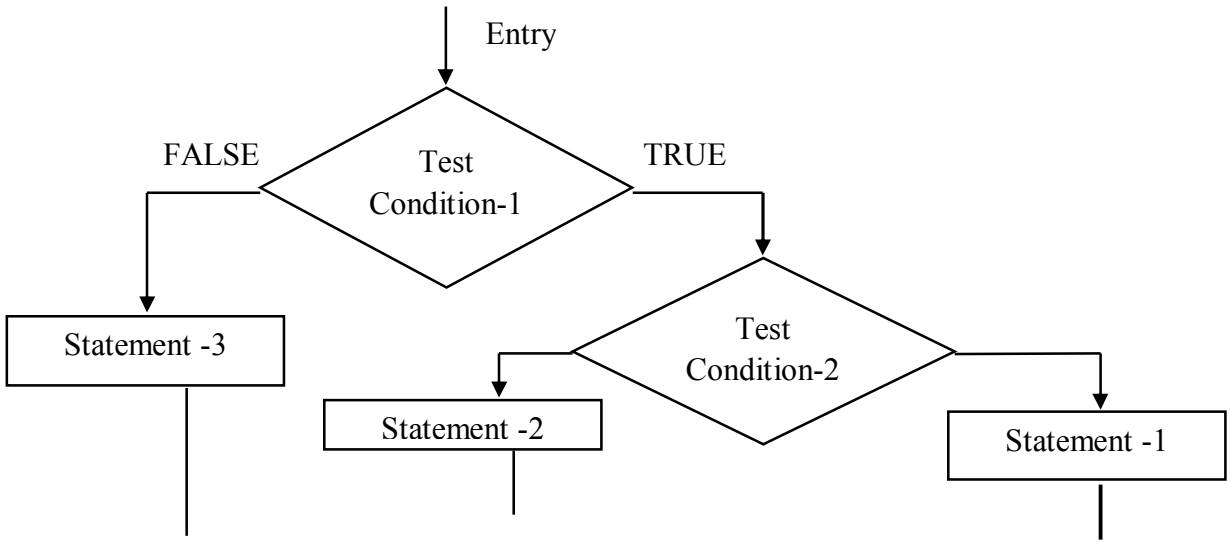
```
if(test expression□1)
{
        if(test expression□2)
        {
                statement□1;
        }
        else
```

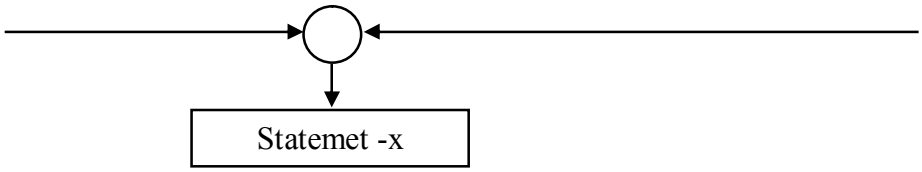
```
                {
                        statement□2;
                }
        }
        else
        {
                statement□3;
        }
        statement –x;
```

➔ Example:

/*WAP to get maximum no. among of given three numbers. */

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b,c;
   clrscr();
   printf("Enter A: ");
   scanf("%d",&a);
   printf("Enter B: ");
   scanf("%d",&b);
   printf("Enter C: ");
   scanf("%d",&c);

  if(a>b)
  {
     if(a>c)
     {
       printf("%d is
 maximum.",a);
     }
    else
     {
       printf("%d is maximum.",c);
     }
  }
  else
  {
     if(b>c)
     {
       printf("%d is
 maximum.",b);
     }
     else
     {
       printf("%d is
 maximum.",c);
     }
  }
  getch();
}
```





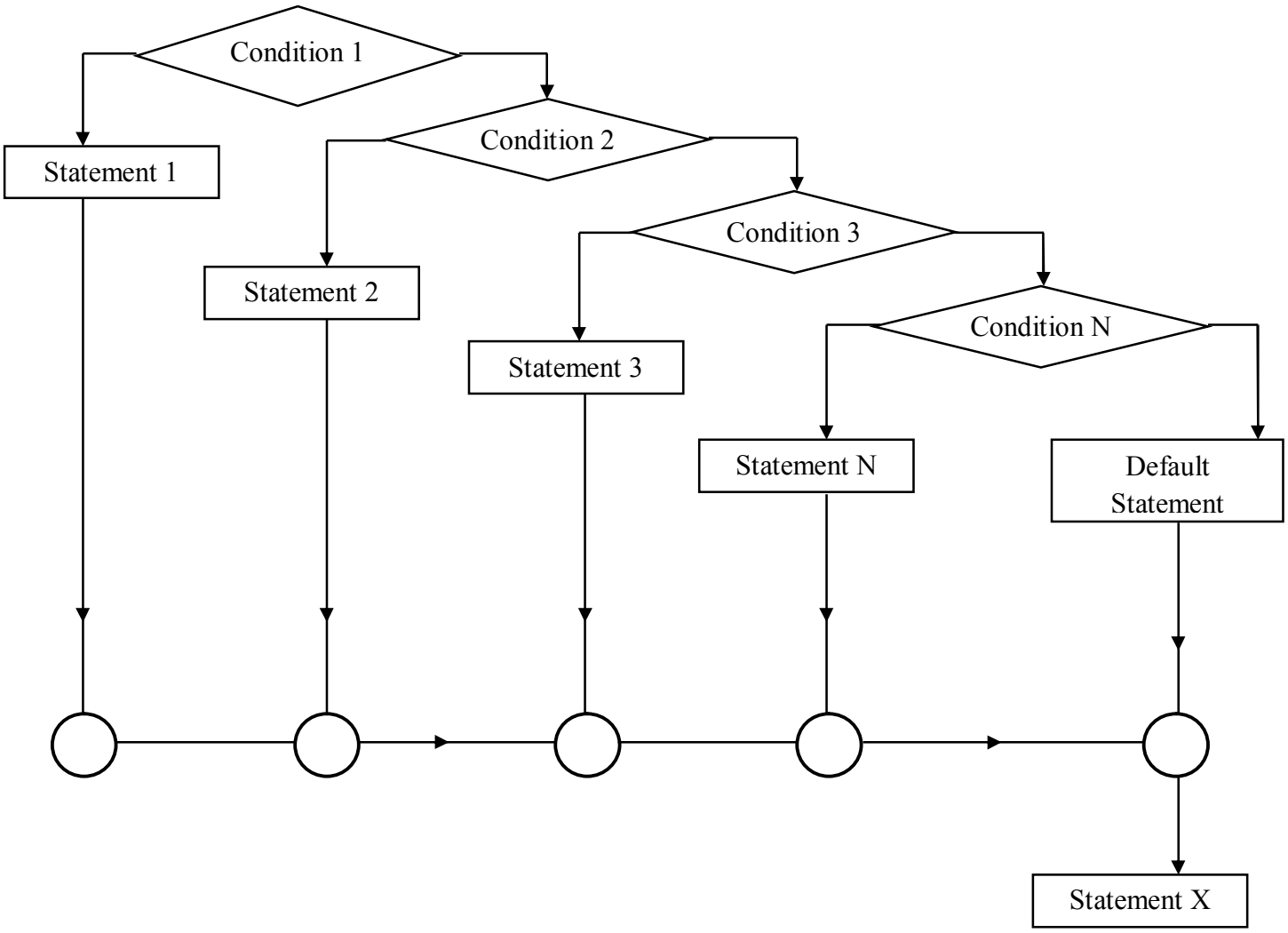
➲ DECISION MAKING WITH ELSE..IF LADDER

➔ Just observe that the program uses the nested if - else.

➔ This leads to three disadvantages:

1) As the No. of conditions are going increased the level of indentation is also increased.
2) An as a result the program is shifted to the right hand side, and readability of your program will be decreased.
3) In case of nested if□else you must take care about the pair of the ifs and elses.
4) You must take care of the opening braces and closing braces of each & every ifs and elses.

➔ There is one more way in which we can write the program of multiple conditions, This involves usage of else if blocks as shown below:



➔ Syntax

```
if(condition 1)
{
        Statement-1;
}
else if (condition 2)
{
        Statement-2;
}
else if(condition 3)
{
        statement - 3;
}
else if (condition n)
{
        Statement-n;
}
else
{
        default-statement;
}
statement-x;
```

➔ Example

```
/*  WAP to find the Student's Grade.  */
#include<stdio.h>
#include<conio.h>
void main()
{
    int sci,maths,eng,tot;
    float per;
  printf("Enter Science Marks: ");
  scanf("%d",&sci);
  printf("Enter Maths Marks: ");
  scanf("%d",&maths);
  printf("Enter English Marks: ");
  scanf("%d",&eng);
  tot=sci+maths+eng;
  per=tot/3;
  if(per>=70)
        printf("\n\tCongratulations.!!! Distinction.");
  else if(per>=60 && per<70)
        printf("\n\tVery Good.!!! First Class.");
  else if(per>=50 && per<60)
```

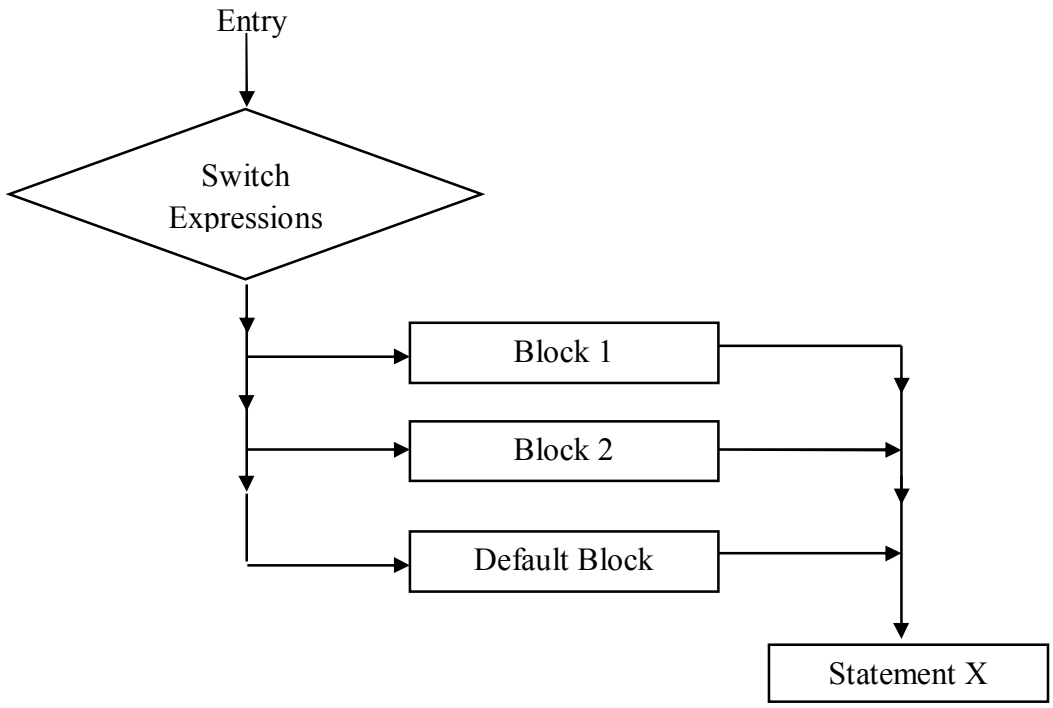
```
        printf("\n\tSecond Class.");
    else if(per>=40 && per<50)
        printf("\n\tPass Class.");
    else
        printf("\n\t***");
}
```

- ➔ As soon as a true condition is found, the statement associated with that condition will be executed, and then the control will transferred to the statement□x.(by skipping the rest of ladders).
- ➔ When all the condition become false, then the else portion that is the final else that contains the default□ statement will be executed, and the control will transferred at the statement□x.
- ➔ That means the program execution will become faster, when you are using the else…if ladder instead of more than one ifs. Because after execution of a particular statement, there is not more conditions are checking, that means skipping the rest of ladder.

B.) switch statement

- ➔ We have already done the decision making using simple ifs and multiple ifs and elses. And as we know that each new form of it was reduced the limitations of previous one.
- ➔ Actually there is no limitation of if…..else ladder.
- ➔ But when the no. of conditions are going to increasing the level of indentation will also going to increasing, at that time even a designer of a program will confuse about the program.
- ➔ So remove this limitation of the else…if ladder, C provides a facility called the Switch□Case statement, also known as "Branching Statement".
- ➔ The syntax of the Switch is as given below:

```
switch(expression)
{
    case value□1:
        block□1;
        break;
    case value□2:
        block□2;
        break;
    …….
    …….
    default:
        default□block;
}
statement x;
```



- ➔ The switch statement is particularly used for menu driven programming.
- ➔ The switch statement checks value of the expression against a list of case value, and when a match is found, a block of statements associated with that case is executed.
- ➔ The expression is an integer or character expressions.
- ➔ value□1, value□2….are the integer constants or the character constants.
- ➔ Each value in the list of case must be unique, that means no 2 case have similar values.
- ➔ Not that, each case keyword with expression is followed by a colon.
- ➔ The block□1, block□2…. Are the statement blocks and may contain 1 or more statement that will be executed.
- ➔ The value of switch (expression) will compared with a list of case values, and when a match is found the statement of that case will be executed.
- ➔ Note that, each block contain the **"break" statement** at the end of each statement block will signals the control at the statement□x which is out of the switch statement.
- ➔ When there is no match found between switch and case value the default case which is optional, and known as else case will be executed.
- ➔ ANSI C permits the use of as many as 257 case labels.
- ➔ **Advantage**

The main advantage of switch statement over if is that it leads to more structured program & the level of indentation is manageable, more so if there are multiple statements within each case of switch.

➔ **Drawback**

Drawback is the use of logical operators. That means the logical operators cannot used in switch..case.

C.) CONDITIONAL OPERATOR

➔ The conditional operator is also known as a "Ternary operator".

➔ It used as an alternative of simple if…else statement.

➔ The combination of ? and : is known as a conditional operator.

➔ Further details We have already learnt in OPERATORS part in 'C' Tokens.

➔ Syntax

Exp1 ? Exp2 : Exp3;

where,, exp1, exp2, exp3 are expressions.

➔ The operator ?: works as follows:

- Exp1 is evaluated first.
- If it is NON-ZERO (TRUE), then the expression Exp2 is evaluated and becomes the value of expression.
- If Exp1 is FALSE, Exp3 is evaluated and its value becomes the value of the expression.
- Note that only one of the expressions (either Exp2 or Exp3) is evaluated.

➔ It replaces simple if…else loop.

➔ Example…

```
if (a > b)
    x = a;          is replaced by      x = (a > b) ? a : b;
else
    x = b;
```

❖ Looping Structures in C

➔ We have already learn about the sequential program and also learn about that how we can control the flow of program and branching of some of statements of the program.

➔ Now we are going to learn that how to perform the particular task no. of times. That means how repeat a task without writing a task no. of times.

➔ This thing can be done using the concepts of loop.

➔ To understand the concept of the loop, it is necessary to understand the concept of the following things:

1) Loop

➔ To perform a particular task no. of times the concepts of loops can be used.

➔ Loops can be of 2 types:

1. **Finite Loop**
   The loop in which the no. of iteration is predefined is known as Finite Loop.
2. **Infinite Loop**
   The loop in which the no. of iteration is not predefined is known as Infinite Loop.

2) Iterations

➔ The meaning of the word "Iteration" is "Repetition".

3) Counter Variable

➔ The variable which is used to count the no. of iterations is known as counter variable.

➔ In looping the sequence of statements are executed until some conditions for termination of the body of the loop are satisfied.

➔ Therefore the program loop is made up of 2 statements:

- The Body of the loop
- The Control statement (A Condition)

➔ Depending on the position of the condition (control statement) in the loop, the loop may be classified into 2 categories:

1. Entry□Controlled loop
2. Exit□Controlled loop

➔ There are 3 methods through which some part of the program can be repeated number of times, they are as given below:

1. The while statement
2. The do…while statement
3. The for statement

1.) THE **while** STATEMENT

➔ The while statement is the simplest of all the loop structure in the C language.

➔ The syntax of the while statement is as given below:

➔ Syntax

```
while( test expression)
{
        body of the loop
}
statement□x;
```

➔ The while statement is known as entry□controlled loop.

➔ Because the condition is checked first and then the body of the loop will be executed if the condition is true.

➔ Here the while is the keyword which denotes that the following block is the loop.

- ➔ If test expression is evaluated to true then and then the body of the loop will be executed, otherwise the control will transfer to the statement□x, which is immediately after the loop.
- ➔ If the condition is true and the loop will executed, then again the condition will be checked if the condition is true than the body of the loop will be executed.
- ➔ This process is continued while the test condition is not evaluated to false.
- ➔ The flow chart and the example of the while statement is as given below:

2.) THE **do…while** STATEMENT

- ➔ The do…while statement is known as an "Exit□Controlled Loop".
- ➔ In case of while loop, the condition is checked first and then the body of the loop will be executed.
- ➔ That means if the condition is not satisfied at first attempt then the control will jump out of the loop.
- ➔ Where as in case of do…while statement the body of the loop is executed first and then the condition will checked, that means the body of the loop is executed if a condition is not satisfied at first attempt.
- ➔ The syntax and example of the do…while statement is as given below:

```
do
{
        body of the loop
}
while (condition);
```

- ➔ In above syntax the do & while are the keywords which defines the loop.
- ➔ Consider the syntax the condition is not checked at the entry of the loop, but the condition is on exiting the loop.
- ➔ According to the syntax, when the control will be reached at the do statement, then the control will proceeds to the body of the loop. And at last the condition will check.
- ➔ Now if the condition is true then the control will go at the do statement again and the body of the loop is executed once again, this process continuous as long as the condition is true.

3.) THE **for** STATEMENT

- ➔ The for statement is the most widely looping control statement among the all statements. This is again an entry controlled looping statement and provides a more concise loop control structure.
- ➔ The for loop is very simple and the syntax of the for loop is something different from the other 2 statements.
- ➔ Syntax

```
for (initialization ; test condition ;increment)
{
```

```
        body of the loop;
    }
```

where,

- ➲ You can see the initialization of the counter variable is also the part of the loop.
- ➲ Here the for statement is consisting of 3 parts:
  - ➿ Initialization (i)
  - ➿ Testing (t)
  - ➿ Increment (i)
- ➲ The initialization of the counter variable is done only once, using an assignment statement.

➔ The second part is consisting of the conditional statement that is the used to control the loop. If you want to know the maximum iterations of the loop, then you can identify it using the conditional expression.

➔ The last part is consisting of the increment or decrement of the counter variable, through which the value of the counter variable is finally reach at the maximum iterations of the loop.

➔ After executing this increment and decrement of the variable the value of the variable the new value of the variable is again tested, if the new value of the variable is satisfy the condition then the body of the loop is again executed. This process is continued till the value of the counter variable is reached at the maximum value of the variable.

➢ Features of **for** loop

1. You can initialize more than one variable at a same time in the for statement.
2. You can give multiple arguments as an increment and decrement part of the for statement.
3. The test condition has any compound relation and the testing need not be limited only to the loop control variable.
4. It is also possible to use the expression to initialize the value of the counter variable.
5. It is also possible to omit one or more sections of the for statement.

➢ Nesting of **for** loop

➔ As an if statement, the for loop may be nested, that means one for loop within another for loop.

➔ This is as given below:

```
for(i=1 ; i<=5 ; i++)  // Outer Loop
{
        □ □ □ □ □ □
        for( j=1 ; j<=5 ; j++)  // Inner Loop
        {
```

□ □ □ □ □ □ □ □ □

}

□ □ □ □ □ □ □

}

➔ The nesting of the loop may continue upto any desired level, but our ANSI C Compiler allows up to 15 levels of the nesting.

➔ The outer loop is for the rows and the inner loop is for the columns.

❖ BREAK STATEMENT

➔ Break statement is useful to exiting prematurely from loop.

➔ Break statement is used to exit from any loop or Switch Case.

- In switch case whenever a break is encountered, it passed the control to the immediately following statement or case
- Break is used to exit from any loop. After Break, the program will start execution from the next sentence which is following the loop. Syntax: **break;**

➔ This statement is used to prematurely stop the execution of the immediately enclosing loop.

➔ Example:

```
void main ()
{
   for (i=1 ;1<=10; i=i+1)
   {
       printf(“  %d”,i);
       if (i==5)
           break;
   }
 }
```

❖ The CONTINUE statement

➔ Syntax: continue;

➔ This statement is used to bypass the remaining statements of the loop and take the control to the beginning of the loop without executing the remaining statements.

➔ Example…

```
void main()
{
        int a;
        for(a=1 ;a<=5;3=a+1)
        {
                printf("%d\n", a);
                if (a==3)
                        continue;
        }
}
```

❖ The GOTO statement

- ➔ The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program.
- ➔ That means if you want to control the flow of control according to a certain condition then you can do it with the help of "goto statement".
- ➔ Actually this statement is referred to as "goto label".
- ➔ C supports the goto statements to branch unconditionally from one point to another in the program.
- ➔ The goto requires a label to identify the place where the branch is to be made.
- ➔ The label is any valid variable name, and must be followed by a colon.
- ➔ The label is placed immediately before the statement where the control is to be transferred.
- ➔ Syntax

```
//Forward Jump
goto label;
        - - - - - - - -
        - - - - - - - -
        - - - - - - - -
        label:
statement;
```

```
//Backward Jump
label:
        statement;
        - - - - - - -
        - - - - - - -
        - - - - - - -
goto label;
```

- ➔ The label can be nay where in the program either before or after the **goto label;** statement.
- ➔ Note that a goto breaks the normal execution of the program.
- ➔ If the **label:** is placed before the statement **goto label; a loop** will be formed and some statements will be executed repeatedly. And such jump is known as **"Backward Jump"**.
- ➔ If the label is placed after the **goto label;** some statements will be skipped & the jump is known as a "**Forward Jump"**.
- ➔ The most common applications of the goto is as given below:
    1. Branching around the statements and group of statements under certain conditions.
    2. Jumping it the end of a loop under the certain conditions, thus bypassing the reminder of the loop during the current pass.
    3. Jumping completely out of the loop under the certain conditions, thus terminating the execution of the loop.

Ch-4
Header Files and Library Functions

- ✓ Importance of Header Files
- ✓ Header Files

❖ Importance of HEADER FILEs

➔ The C language is accompanied by a number of library functions.
➔ ANSI C committee has standardized header files which contain these functions.
➔ Basically header files are the internal built-in function files which are useful for creating any C program.
➔ The header files also contain other information related to the use of the library functions, such as symbolic constant definitions.

➔ Header files are included in C program by using **#include**.
➔ C provides built-in functions that are defined in files with **.h** extension.
➔ In order to execute functions defined in these header files it is necessary to include then in the beginning of the program.

❖ HEADER FILES

➔ Some of the popular header files are…
- stdio.h
- conio.h
- math.h
- string.h
- ctype.h

stdio.h
- ✓ A stream is a source / destination of data that may be associated with a desk or other peripheral.
- ✓ character testing & conversion functions.
- ✓ Functions are as below:

clearerr();	fclose();	feof();	ferror();
fflush();	fgetpos();	fopen();	fread();
freopen();	fseek();	fsetpos();	ftell();
fwrite();	remove();	rename();	rewind();
setbuf();	setvbuf();	tmpfile();	tmpnam();
fprintf();	fscanf();	printf();	scanf();
sprintf();	sscanf();	vfprintf();	vprintf();
vsprintf();	fgetc();	fgets();	fputc();
fputs();	getc();	getchar();	gets();
putc();	putchar();	puts();	ungetc();
perror();			

conio.h
- ✓ Console input/output header file.

- ✓ Functions are below:

No.	Functions	Data type Returned	Task
1	getch()	int	Gets a character from console but doesn't echo to screen.
2	getche()	int	Gets a character from console and echoes to the screen.
3	getchar()	int	Gets a character from standard input device.
4	clrscr()	void	Clears text mode window.
5	goto xy()	int	Positions the cursor in text window.
6	cprintf()		Sends formatted output to the text window on the screen.
7	textcolor()	int	Selects a new character color in text mode.
8	textbackground()	int	Selects a new text background color.

- math.h
  - ✓ The header file math.h declares mathematical functions and macros.
  - ✓ The math.h header file is following function included in it:
  - ✓ Functions are as below:

abs();	acos();	asin();	atan();
cos();	exp();	fabs();	floor();
fmod();	frexp();	log();	mod();
pow();	sin();	sqrt();	tan();
ceil();			

- string.h
  - ✓ This header file is used to perform task on character data or for string value.
  - ✓ Functions are as below:

memchr();	memcmp();	memcpy();	memmove();	memset(0;
strcat();	strncat();	strchr();	strcmp();	strncmp();
strcoll();	strcpy();	strncpy();	strcspn();	strerror();
strlen();	strpbrk();	strrchr();	strspn();	strstr();
strtok();	strxfrm();	strupr();	strlwr();	strrev();

- ctype.h
  - ✓ This header file is used to declares functions for testing characters.
  - ✓ By using this header file we can perform TESTING and CONVERSION characters.
  - ✓ Functions are as follow:

isalnum();	isalpha();	iscntrl();	isdigit();	isgraph();
islower();	isprint();	ispunct();	isspace();	isupper();
isxdigit();	tolower();	toupper();		

Ch-5

User Defined Functions

- ✓ Introduction
- ✓ Function Definition
- ✓ Passing & Returning Values to Function
- ✓ Types of UDF
- ✓ Recursion
- ✓ Pointers
- ✓ Storage Classes
- ✓ Creating of Library

❖ INTRODUCTION

➔ The C language functions can be categories…

1. Library Functions
2. User Defined Functions

➔ The main difference between Library and UDF is…

1. Library Function:
   - Not required to be written by the user.
   - Ex. printf(), scanf() etc…
2. UDF
   - Has to be developed by the user at the time of writing a program.
   - However, a UDF can late become a part of the C Program Library.
   - **main() is UDF.**

❖ FUNCTION DEFINITION

➔ A function is a…

- block of code
- in sequential manner
- to perform a particular task.

➔ Every C program is collection of functions.

➔ Syntax…

```
function_name(argument list)
{
        variable declarations;
        function statements;
        ----------------------
        ----------------------
        return (expression);
}
```

❖ Passing & Returning Values to Function

➔ Arguments to a function are usually passed in two ways:

1. Call by Value.
2. Call by Reference.

➔ That means the functions which are created by the user are known as user defined functions.

➔ The user defined functions are declared, defined and called within the c file as per the requirement of the user.

➔ There are some points that can be noted out as given below:

1) The Function is defined outside the main ( ).
2) The Function can be called from the main() or any other UDF.
3) After executing each of the UDF the control will transfer in the main( ).
4) The function call statement is followed by () and semincolon (;).
5) The Function definition is followed by the ( ).
6) There is no need to specify the name of the variable at the time of prototype of the function.
7) One C program can have more than one UDF.
8) Each and every program has one UDF that is obviously main( ).
9) The functions are executed in order, in which they are called, not in which they are defined.

➔ If you want to understand UDF then you have to understand 3 part of UDF:

- ⇨ Declaration (Prototype) of Function
- ⇨ Calling of Function
- ⇨ Definition of Function

1.) Declaration (Prototype) of Function
   - ✓ Before use function, we have to declare it.
   - ✓ It is optional part of UDF.
   - ✓ Function declaration also follows rules of variable naming.
   - ✓ Syntax: return_type function_name(argument list);

2.) Calling of Function
   - ✓ It is one of the executable statement of C-program.
   - ✓ We can call UDF same like calling other library function.
   - ✓ Syntax: function_name(argument list);

3.) Definition of Function
   - ✓ Programmer writes logic of function in this part.
   - ✓ Once programmer defines it, you can call it as many time as you want.
   - ✓ Syntax:

```
return_type function_name(argument list)
{
        // declaration statements;
        // executable statements;
        // return statement;
}
```

❖ Types of UDF

1) Function without argument without return value.
2) Function with argument without return value.
3) Function with argument with return value.
4) Function without argument with return value.

1) Function without argument without return value

➔ When the function has no argument, it does not receive any data from the calling function.

➔ And when the function does not return any value, the calling function does not receive any data.

➔ Let's take one example…

```
/* W.A.P. to add by using UDF concept. */
    #include<stdio.h>
    #include<conio.h>
    void add(void) // Function prototype
    void main( )
    {
            clrscr( );
            add( ); //Function calling statement
            getch( );
    }
    void add( ) // Function definition
    {
            int a,b,c;
            printf( "Enter a : ");
            scanf("%d",&a);
            printf( "Enter b : ");
            scanf("%d",&b);
            c=a+b;
            printf("\n Addition is : %d",c);
    }
```

2) Function with argument without return value

➔ In this category, the calling function has some data that can be passed as an argument.

➔ But the called function has no data so that it can't return any value to the calling function.

➔ That means one way communication is possible between functions.

➔ Example

```
    #include<stdio.h>
    #include<conio.h>
    void add(int ,int)
    void main( )
    {
            int a,b;
            printf("\n Enter a:");
            scanf("%d",&a);
```

```
        printf("\n Enter b:");
        scanf("%d",&b);
        add(a,b); //function call with 2 arguments…(formal argument)
        getch( );
}
void add( int a, int b)
{
        int c;
        c=a+b;
        printf("\n Addition is : %d",c);
}
```

3) Function with argument with return value

- ➔ In this type the calling function has some data that can be passed as an argument.
- ➔ And the called function also response the calling functions, that means the called function can return a value to the calling function by using the return statement.
- ➔ Return is the keyword which can be used to return the values from the functions.
- ➔ Example

```
#include<stdio.h>
#include<conio.h>
int add( int,int)
void main( )
{
        int a,b,c;
        clrscr( );
        printf("\n Enter a:");
        scanf("%d",&a);
        printf("\n Enter b:");
        scanf("%d",&b);
        c=add(a,b);
        printf("\n The Addition is %d",c);
}
int add(int a, int b)
{
        int c;
        c=a+b;
        return (c);
}
```

4) The Function without argument with return value

- ➔ In this type the calling function cannot send any argument to the called functions as an argument.
- ➔ But the called function can send the value by using a return statement as a response.
- ➔ This is one type of again a "One way Communication".
- ➔ Example

```
#incclude<stdio.h>
#include<conio.h>
int add(void)
{
        int c;
        c=add();
        getch( );
}
int add( )
{
        int a,b,c;
        printf("\n Enter a: ");
        scanf("%d",&a);
        printf("\n Enter b: ");
        scanf("%d",&b);
        c=a+b;
        return ( c );
}
```

❖ RECURSION

- ➔ Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.
- ➔ The process is used for repetitive computation in which each action is stated on forms of previous result.
- ➔ The process is used for repetitive computation in which each action in terms of a previous result.
- ➔ Many iterative problems can be written in this form.
- ➔ In order to solve a problem recursively, two condition must be satisfied.
- ➔ First, the problem must be written in recursive form and second the problem example, we wish to calculate the factorial of a positive integer quantity.
- ➔ We would normally express this problem as n1=1 x 2 x 3 x 4 … x n where n is the specified positive integer.
- ➔ However, we can also express in another way, by writing n=1 = n*(n-1).

- ➔ This is a recursive statement of the problem, in which last expression provides a stopping condition for the recursion.
- ➔ When a recursive program is executed the recursive function calls are not executed immediately.
- ➔ Rather, they are placed on a stack until the condition that terminates the recursion is encountered.
- ➔ The function calls are then executed in reverse order, as they are popped off the stack.
- ➔ If a recursive function contains local variables, a different set of local variables will be created during each call.
- ➔ The name of the local variables, will be cause always be the same, as declared within the function.
- ➔ However, the variables will represent a different set of values each time the function is executed.
- ➔ Each set of values will be stored on the stack, so that they will be available as the recursive process "unwinds" i.e. as the various function calls are "popped" off the stack and executed.
- ➔ Example…

/* W.A.P. to find factorial using the concept of RECURSION. */

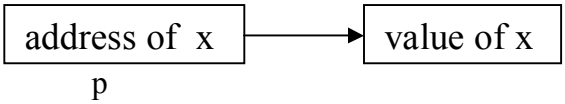
```
#include<stdio.h>
#include<conio.h>
int fact(int n)
{
        int f;
        if(n==1)
                return(1);
        else
                f=n*fact(n-1);
        return(f);
}
void main( )
{
        int n,m;
        clrscr( );
        printf("\n\tEnter any no.: ");
        scanf("%d",&n);
        m=fact(n);
        printf("\n\tFactorial is %d",m);
        getch( );
```

}

❖ POINTERS

➔ A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers have a number of useful applications.

➔ Pointer is one kind of variable in which you can store memory information (address) of its related another variable.



➔ Steps to declare pointer are as follows:

- ⇨ Declare base variable int x;
- ⇨ Declare pointer type variable related to base variable int *p;
- ⇨ Establish relation b/w. base and pointer variable p = &x;

➔ The unary operators & and * is called ADDRESS and INDIRECTION operator to refer address of that variable and to indicate variable as pointer variable respectively.

➔ Example…

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int x=10,y;
        int *ptr;
        ptr=&x;
        y=*ptr;
        printf("\n\t  X: %d", x);
        printf("\n\t %d refers %d", &x, x);
        printf("\n\t %d is stored at %d",*&x, &x);
        printf("\n\t %d refered with %d", *ptr, &ptr);
        printf("\n\t Y: %d", y);

        *ptr=25;
        printf("\n\t Now X=%d", x);
        getch();
}
```

➔ The advantages of using pointers are:

1. Function can't return more than one values but it can modify many pointer variables and can return more than one variable.

2. In the case of ARRAYS, We can decide the N size array @ runtime by allocating necessary space.
3. Main advantage: **DYNAMIC MEMORY ALLOCATION.**

➔ The disadvantages of using pointers are:

- Program may crash during runtime for the storage of pointers.

➢ Passing value to Function

➔ Function can be call in following two ways:

1. Call by Value (Pass by Value)
2. Call by Reference (Pass by Reference)

➔ Explanation…

1.) Call by Value (Pass by Value)

- ✓ This method copies the *value of* an argument into the formal parameter of the subroutine.
- ✓ In this case, changes made to the parameter have no effect on the argument.
- ✓ By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.
- ✓ Consider the following program:

```
#include <stdio.h>
#include<conio.h>
int sqr(int x);

int main(void)
{
    int t=10;
    printf("%d is the square of %d", sqr(t), t);
    return 0;
}
int sqr(int x)
{
    x = x*x;
    return(x);
}
```

2.) Call by Reference (Pass by Reference)

- ✓ In this method, the *address* of an argument is copied into the parameter.
- ✓ Inside the subroutine, the address is used to access the actual argument used in the call.
- ✓ This means that changes made to the parameter affect the argument.
- ✓ Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function.

- ✓ Pointers are passed to functions just like any other value.
- ✓ Of course, you need to declare the parameters as pointer types.
- ✓ Example…

```
void swap(int *,int *);
void main()
{
    int i, j;
    i = 10;
    j = 20;
    swap(&i, &j); /* pass the addresses of i and j */
    printf("\n\tAfter swapping…");
    printf("\n\t\t  i: %d", i);
    printf("\n\t\t  j: %d", j);
    getch();
}
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put x into y */
}
```

Note:

C++ allows you to fully automate a call by reference through the use of reference parameters.

❖ STORAGE CLASSES

- ➔ In 'C' all the variables have data type and storage classes.
- ➔ It explains the lifetime of the storage associated with the variable.
- ➔ Variables tell 4 things…
  - Storage area of variable,
  - Initial value of variable,
  - Scope of variable,
  - Life of variable.

Class / Properties	Automatic	Register	Static	External
Definition	A variable declared without any storage class is considered to be an automatic variable.	A value stored in the CPU register can always be accessed faster than the one which is stored in memory.	The variable is initialized only once for the program. Irrespective of the multiple function	External variables are declared outside all functions, yet are available to all function that wants to use them.

			calls the value persists between multiple calls to function.	
Storage	memory	CPU registers	memory	memory
Default initial value	garbage value if not initialized	garbage value if not initialized	Zero (Null)	Zero
Scope	only in the block where it is defined	only in the block where it is defined	only in the block where it is defined	Global
Life	till the block is executing.	till the block is executing.	the variable is not destroyed after the block or function is over.	As long as the program's execution does not end.
Keyword	auto	register	static	extern
Syntax	auto data_type variable; OR data_type variable;	register data_type variable;	static data_type variable;	extern data_type variable;
Example	auto int a OR int a;	register int a;	static int a;	extern int a;

❖ Creating of Memory

- ➔ We can add UDF to the library.
- ➔ When we use these UDF in other applications as a library function then we can save compile time because we have already functions as in compiled form.
- ➔ We can use these functions as built-in functions in 'C'.
- ➔ Let us see an example to make SQUARE function as library file.

- ➔ Example…

```
/* square.c */
int square(int val);
{
        int temp;
        temp = val*val;
        return temp;
}
```

- ➔ After the compilation of **square.c** , 'C' compiler will create a file **square.obj** which will contain all the codes in machine language.
- ➔ Now add this function to **MATH.H** by using command…

```
C:\>tlib maths.lib + c:\square.obj
```

➔ Now we can use square function as a library file.

➔ Example…

```
#include<stdio.h>
#include<conio.h>
#include "c:\square.h"

void main()
{
        int num,sqr;
        clrscr();
        printf("Enter any number to find its square: ");
        scanf("%d" , &num);
        sqr=square(num);

        printf("Square of %d is %d" , num, sqr);
        getch();
}
```

Ch-6

ARRAY

- ✓ Introduction
- ✓ Single Dimensional Arrays
- ✓ Two- Dimensional Arrays
- ✓ Initialization and working with Array
- ✓ Passing Array Elements to Function
- ✓ Use of Pointers in Array
- ✓ Sorting Numeric and String Arrays
- ✓ String Function and Operations

❖ Introduction

- ➔ Up to this chapter, we are learning to store only one value in the variable at a time.
- ➔ Now suppose you want to store multiple values in the program, at that time you need not to declare that much variables in your program. C supports the concept of an array.
- ➔ Using an array, we can store multiple values within one variable.
- ➔ The concept of an array is used to handle the large amount of data.

- ➔ **ARRAY:: An Array is a collection of the elements having similar data type.**
- ➔ We can use an array to represent not only simple lists of values but also tables of data in two or three or more dimensions.
- ➔ There are 3 types of an array available in C.
    A. One dimensional Array
    B. Two dimensional Array
    C. Multi dimensional Array
- ➔ It is simple a grouping of like data type.
- ➔ It is one of the DATA STRUCTURE of C.
- ➔ It is also known as DERIVED DATA TYPE in C.

❖ One Dimensional Array

- ➔ The array is said to be an one dimensional, if the array variable name has only one subscript.
- ➔ That means a list of items can be given one variable name using only one subscript and such variable us called a "Single scripted variable" or "One dimensional Array".
- ➔ Syntax

    data_type variable_name [value];

    where,

    The data type is any valid c data type.
    Variable is any valid c variable.
    And
    The value specifies that how many values can be stored inside the variable.

➲ Declaration of a one dimensional Array

int a[5];

➲ The computer reserves the five locations as shown below:

a[0], a[1], a[2], a[3], a[4]

➲ Explanation..

- ➔ Declares the variable named a and can hold 5 values at a same time.
- ➔ All the values may belong to only one data type that is integer.
- ➔ **Note**
  - ✓ Any references to the arrays outside the declared limits would not necessarily cause an error. Rather it might result in unpredictable results.
  - ✓ The size should be either a numeric constant or a symbolic constant.
- ➔ Example

```
//A Program of the single dimensional array.....
    #include<stdio.h>
    #include<conio.h>
    void main( )
    {
        int a[5] ,i;
        for(i=0;i<=5;i++)
        {
            scanf("%d",&a[i]);
        }
        for(i=0;i<=5;i++)
        {
            printf("\n%d",a[i]);
        }
        getch();
    }
```

❖ Two Dimensional Array

- ➔ We all know that an array variable can store a list of values.
- ➔ Now, if we want to store a list of value in form of table, at that time two□ dimensional array is used.
- ➔ The two dimensional array is just a variable name with 2 subscript.
- ➔ The same rule is applied over here. That is the subscript is start from 0 and the last subscript is size□1.
- ➔ Here we have to specify to things in 2□D array.
  1. No. of rows
  2. No. of columns
- ➔ Syntax

```
data_type variable_name [row size] [column size];
```

- ➔ Example

```
int No[3][4];
```

- ➔ The above variable no can hold total 12 values.
- ➔ The memory representation of the no variable is as given below:

  No[0][0] No[0][1] No[0][2] No[0][3]
  No[1][0] No[1][1] No[1][2] No[1][3]
  No[2][0] No[2][1] No[2][2] No[2][3]
- ➔ There is another concept of 2□D character array.
- ➔ That means you can store only one string in 1□D array of the character data type.
- ➔ But when you want to store more than one string within one variable at that time 2□D array of the character data type is used.

➲ To Store more than one string within 1 variable

```
char nm [3][4];
```

⇨ The variable 'nm' can store 3 strings, each having 4 characters.

➲ Remember

- ➔ There are 2 ways of getting string form the key board.
  - ➿ Either you use scanf( ) function.
  - ➿ Or you use the gets ( ) function.
- ➔ But remember that the scanf( ) function can't accept the space within a string.
- ➔ And the gets ( ) function can accept the space within a string.
- ➔ When we want to create a variable that can hold the values of different data types, at that time the concept of structure variable.

➔ Example…

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

    for(i=0;i<=2;i++)
```

```
    {
        for(j=0;j<=2;j++)
        {
            printf("%3d",a[i][j]);
        }
        printf("\n");
    }

    getch();
}
```

❖ Initialization & Working with ARRAY

➔ There are two ways to initialize an array:
1. At compile time
2. At run time.

➔ Array can be initialized by two stages:
1. Declare an array with its data-type.
2. Assign values into array of its data-type.

➢ Compile Time Initialization

➔ You can initialize an array at compile time by defining a set of values within {…….}.

➔ Syntax

```
data_type variable_name[size]={list of values};
```

➔ Size may be omitted in such case.

➔ Character arrays can be initialized in a same way.

➔ If you enter less value as per declared size, in this case elements are initialized to ZERO, if array is INT and to NULL, if array is CHAR.

➢ Run Time Initialization

➔ In this type of array, size of array can be declared at the time of execution of program.

➔ Also elements must be declared at the time of execution.

➔ Here, compiler does not come in focus for memory allocation for array direcftly.

➔ Example…

```
int arr[5];
printf("Enter 5 numbers: ");
scanf("%d %d %d %d %d", &arr[0],&arr[1],&arr[2],&arr[3],&arr[4]);
```

❖ PASSING ARRAY ELEMENS TO FUNCTION

- ➔ An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function.
- ➔ The manner in which the array is passed differs mainly.
- ➔ However, from that of an ordinary variable.
- ➔ To pass an array to a function, the array mane must appear by itself without brackets or subscripts as an actual argument within the function call.
- ➔ The corresponding format argument is written in the same manner, though it must be declared as an array within the formal argument declaration.
- ➔ When declaring array as a format argument, the array name is written with a pair of empty square brackets.
- ➔ The size of the array is not specified within the format argument declaration.
- ➔ Example…

```
int arsum(int * start_here)
{
      int ndays = 0;
      while (1)
      {
            ndays += *start_here++;
            if (*start_here == 0) break;
      }
      return ndays;
 }
void main()
{
       int daze[13] = {31,28,31,30,31,30,31,31,30,31,30,31,0};
       int days_in_array;

       /* days_in_array =  arsum(&daze[0]); */
       days_in_array =  arsum(daze);

       printf("The year is %d days long\n",days_in_array);
       getch();
 }
```

❖ Use of Pointers in Array

- ➔ Pointers, of course, can be "pointed at" any type of data object, including arrays.
- ➔ It is important to expand on how we do this when it comes to multi-dimensional arrays.

```
int *ptr;
ptr = &my_array[0];       /* point our pointer at the first integer in our array */
```

- ➔ As we stated there, the type of the pointer variable must match the type of the first element of the array.
- ➔ In addition, we can use a pointer as a formal parameter of a function which is designed to manipulate an array. e.g.
- ➔ Given:

```
int array[3] = {1, 5, 7};
void a_func(int *p);
```

- ➔ Some programmers might prefer to write the function prototype as:

```
void a_func(int p[]);
```

- ➔ which would tend to inform others who might use this function that the function is designed to manipulate the elements of an array.

❖ Sorting Numeric and String Arrays

Numeric Array Sorting

- ➔ There are many different cases in which sorting an array can be useful.
- ➔ Algorithms can often be made simpler and/or more efficient when the input data is sorted.
- ➔ Furthermore, sorting is often useful for human readability, such as when printing a list of names in alphabetical order.
- ➔ Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some criteria.
- ➔ The order in which these elements are compared differs depending on which sorting algorithm is used, and the criteria depends on how the list will be sorted (ascending or descending order).
- ➔ Example…

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j,a[10],temp;
        clrscr();

        for(i=0;i<=9;i++)
        {
                printf("Enter your No.a[%d]= ",i);
                scanf("%d",&a[i]);
        }
```

```
        for(i=0;i<=9;i++)
        {
                for(j=i+1;j<=9;j++)
                {
                        if(a[i]>a[j])
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
                }
        }
        printf("\nAfter sorting. . .");

        for(i=0;i<=9;i++)
        {
                printf("\n%d",a[i]);
        }

        getch();
}
```

String Array Sorting

- A string is a group of characters, usually letters of alphabets.
- To format your output in such a way that it looks perfect, you need an ability to format it in proper manner.
- A complete definition is a series of "char" type data terminated by NULL character, is a ZERO.
- Example…

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j,k;
        char *arr[3]={"Jay", "Shree", "Krishna");
        char *temp;
        clrscr( );

        printf("\n\n Entered string are…");
        for(i=0;i<5;i++)
```

```
        {
                printf("%s\n" , arr[i]);
        }
        for(j=0;j<i-1;j--)
        {
                for(k=j+1;k<I;k++)
                {
                        if( strcmp ( arr[j],arr[k] ) > 0 )
                        {
                                strcpy(temp, arr[j]);
                                strcpy( arr[j], arr[k]);
                                strcpy( arr[k], temp);
                        }
                }
        }
        printf("\n Sorted in Ascending order \n");
        for(i=0;i<5;i++)
        {
                printf("\n\t %s", arr[i]);
        }
        getch();
}
```

❖ String Function and Operations

- ➔ A string is an array of characters whose last character is \0. A typical string, such as Pacifique, is graphically represented as follows

P	a	c	i	f	i	q	u	e	\0

- ➔ The last character \0 is called the null-terminating character.
- ➔ For this reason, a string is said to be null-terminated.
- ➔ The string library ships with a lot of functions used to perform almost any type of operation on almost any kind of string.
- ➔ Used under different circumstances, the string functions also have different syntaxes.

The strlen() Function

- ➔ In many operations, you will want to know how many characters a string consists of. To find the number of characters of a string, use the **strlen()** function.
- ➔ Its syntax is:

```
int strlen(const char* Value);
```

➔ The strlen() function takes one argument, which is the string you are considering. The function returns the number of characters of the string.

➔ Here is an example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  char *School = "Manchester United";
  int Length = strlen(School);
  clrscr();
  printf("\n\tThe Length of  \"%s\" is %d characters.", School, Length);
  getch();

}
```

➔ Output: The Length of "Manchester United" is 17 characters.

The strcat() Function

➔ If you have two strings, to append one to another, use the strcat() function.

➔ Its syntax is:

```
char *strcat(char *Destination, const char *Source);
```

- ✓ The strcat() function takes two arguments.
- ✓ The second argument, called the source string, is the string you want to add to the first string; this first string is referred to as the destination.
- ✓ Although the function takes two arguments. It really ends up changing the destination string by appending the second string at the end of the first string.

➔ This could be used to add two strings.

➔ Here is an example…

```
#include <stdio.h>
#include <conio.h>
void  main()
{
    char *Make = "Ford ";
    char *Model = "Explorer";
    printf("\n\t Originally, Make = %s", Make);
    strcat(Make, Model);
      printf("\n\n\t After concatenating, Make = %s" ,Make);
    getch();
}
```

This would produce:
Originally, Make = Ford
After concatenating, Make = Ford Explorer

The strcpy() Function

➔ The strcpy() function is used to copy one string into another string.
➔ In English, it is used to replace one string with another.
➔ The syntax of the strcpy() function is:

```
char* strcpy(char* Destination, const char* Source);
```

- ✓ This function takes two arguments.
- ✓ The first argument is the string that you are trying to replace.
- ✓ The second argument is the new string that you want to replace.
- ✓ There are two main scenarios suitable for the strcpy() function:
  - ➲ To replace an existing string or to initialize a string.

```
#include <stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        char carName1[ ] = "Ford Escort";
        char carName2[ ] = "Toyota 4-Runner";
        clrscr();
        printf("\n\nThe String Copy Operation");
        printf("\n\tFirst Car: %s", carName1);
        printf("\n\tSecond Car: %s" ,carName2);
        strcpy(carName2, carName1);
        printf("\n\nAfter using strcpy()...");
        printf("\n\tFirst Car: %s", carName1);
        printf("\n\tSecond Car: %s", carName2);
        getch();
}
```

This would produce:

```
The String Copy Operation
    First Car: Ford Escort
    Second Car: Toyota 4-Runner
After using strcpy()...
    First Car: Ford Escort
    Second Car: Ford Escort
```

The strcmp() Function

➔he strcmp() function compares two strings and returns an integer as a result of its comparison.

➔Its syntax is:

int strcmp(const char* S1, const char* S2);

This function takes two strings, S1 and S2 and compares them. It returns …

- ✓ A negative value if S1 is less than S2
- ✓ Zero if S1 and S2 are equal
- ✓ A positive value if S1 is greater than S2

```
#include <stdio.h>
#include <conio.h>
void main()
{
        char *FirstName1 = "Andy";
        char *FirstName2 = "Charles";
        char *LastName1 = "Stanley";
        char *LastName2 = "Stanley";
        clrscr();

        int Value1 = strcmp(FirstName1, FirstName2);
        int Value2 = strcmp(FirstName2, FirstName1);
        int Value3 = strcmp(LastName1, LastName2);

        printf("\n\t The result of comparing: %s and %s is %d", FirstName1, FirstName2,
                        Value1);

        printf("\n\t The result of comparing: %s and %s is %d", FirstName2, FirstName1,
                        Value2);

        printf("\n\t The result of comparing: %s and %s is %d", LastName1, LastName2,
                        Value3);

        getch();
}
```

This would produce:

```
The result of comparing Andy and Charles is –2
The result of comparing Charles and Andy is 2
The result of comparing Stanley and Stanley is 0
```

Ch-7

Structure

- ✓ Introduction
- ✓ Declaration and Initialization of Structures
- ✓ Accessing Structure Members
- ✓ Memory Allocation
- ✓ Nested Structure
- ✓ Arrays of Structure
- ✓ User Defined Data Types
- ✓ Pointers of Structure
- ✓ Structure and Functions
- ✓ Unions

❖ Introduction

- ➔ Structure is the collection of elements having different data types.
- ➔ A structure is a collection of one or more variables types grouped under a single name for easy manipulation.
- ➔ The variables in a structure, unlike those in an array, can be of different variable types.
- ➔ A structure can contain any of C's data types, including arrays and other structures.
- ➔ Each variable within a structure is called a member of the structure.

❖ Declaration and Initialization of Structures

- ➔ Syntax

```
struct tag_name
{
        data type member..1;
        data type member..2;
        data type member..3;
        □□□□□□□□□□□□□
        □□□□□□□□□□□□□
        data type member..n;
};
struct tag_name struct type variable;
```

where,

- ➔ The struct keyword defines the structure.
- ➔ The tag name is the identifier of the structure. This should be meaningful.
- ➔ Member..1,member..2,member..3 are the members of the structure.
- ➔ The template of the structure must be enclosed within a curly braces {…} and terminated by a semicolon (;).
- ➔ And in the last you can see one variable which is not belongs to a particular data type. It's data type is structure.
- ➔ The structure is the user defined data type. It is not a built in data type.

Remember one point

Internally the members of the structure have not any space in the memory at compile time, when they are associated with the structure type variable at that time they have been given a memory.

Let see on example that contain the data of student.

➔ Example

```
// A program of the structure….
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct stud
    {
            int rno;
            char nm[15];
    };
    struct stud s;
        printf("\n Enter your Roll No. :");
        scanf("%d",&s.rno);
        printf("\n Enter your Name :");
        scanf("%s",&nm);
        printf("\n Roll No. : ",rno);
        printf("\n Name : ",s.nm);
        getch( );
}
```

➔ In the above program you can see that the structure has 2 members.
➔ And can be access by using a structure type variable and member operator.

❖ Accessing Structure Members

➔ To access the members of the structure the member operator (.) is used.
➔ Syntax:

structure_name . membername

➔ Example:

s1.rollno, e1.empno etc…

❖ Memory Allocation

➔ Memory Allocation of structure in C is same as array.
➔ Compiler allocates continuous memory to structure elements in one bunch of memory.
➔ But remember that…
  - Memory allocation is done only at the time of declaration of variable of structure type not at the time of structure declaration.

➔ We have talk about COMPILE TIME ALLOCATION yet.

➔ Now, Let us see about RUN TIME ALLOCATION…

- There are some library functions for allocating the memory during the execution of the program.
- These functions are known as MEMORY MANAGEMENT FUNCTIONS.

➢ Memory Management Functions

1. malloc()
2. calloc()
3. free()

1.) malloc()

- ✓ It reserves a block of memory for specified size.
- ✓ It returns a pointer type of VOID.
- ✓ So it can be assigned for any type of pointer.
- ✓ Syntax:

```
pointer = (pointer type*) malloc (byte size);
```

- ✓ Example:

```
ptr = (int*) malloc (10 * sizeof(int));
```

2.) calloc()

- ✓ It is normally used for requesting runtime memory space for storing ARRAYS, STRUCTURES, UNIONS, etc…
- ✓ It allocates multiple blocks of memory storage and each of block are of same size.
- ✓ Syntax:

```
pointer = (data type*) calloc (n, element size);
```

- ✓ Example:

```
ptr = (float*) calloc (20, sizeof(float));
```

Here,

- ➲ calloc allocates 4 bytes to hold data for 20 records of float type.
- ➲ When calloc is used, we must be sure that the requested memory has been allocated successfully.

3.) free()

- ✓ The function is used for releasing memory spaces used in program.
- ✓ When memory spaces are not in use, it is programmer's responsibility to free those spaces when not needed.
- ✓ Syntax:

```
free(pointer);
```

- ✓ Example:

```
free(ptr);
```

❖ Nested Structure

➔ Nested structures are the structures within structure.

➔ We can create a new structure inside the old structure; the new structure will be called NESTED STRUCTURE.

➔ Example…

```
struct stud
{
    char *name;
    int rno;
    float fees;

    struct result
    {
        int mark[5],sum;
        float per;
    };
};
```

Here, STUD is the OLD STRUCTURE; RESULT is the NESTED STRUCTURE.

❖ Arrays of Structure

➔ Arrays can be used to assign structure member and its value comfortably.

➔ It can be declared, then each elements of array represents a structure variable.

➔ Example:

```
struct College
{
    int clgid, clgstars, employee, courses, branches;
    char *name;
};
main()
{
    typedef struct College;
    College CLG[3];
}
```

❖ User Defined Data Types

➔ 'C' allows to define new data types equivalent to the existing system data types using TYPEDEF statement.

➔ Example:

```
typedef struct
{
        int  dd;
        int  mm;
        int  yyyy;
}DOB;
```

Here, we can now use DOB as built-in data type in another structure.

❖ Pointers to Structure

➔ It is just like the pointers to built-in data type's variables.

➔ Example:

```
struct stud
{
    char *name;
    int roll;
    int std;
    float per;
}*s1;
```

❖ Structure and Functions

➔ Structures may be passed to function either by value or by reference.

➔ Mostly, Call by reference is used.

➔ Example:

```
#include <stdio.h>
#include <conio.h>
struct square
{
    int val;
}sqr;

void main()
{
    int ans,squr();
    clrscr();
    printf("Enter any value: ");
    scanf("%d", &sqr.val);
    ans=squr(&sqr);
```

```
    printf("Square of %d is %d", sqr.val, ans);
    getch();
}
int squr(struct square *s)
{
    return (s→ val * s→val);
}
```

❖ Unions

- ➔ A union, is a collection of variables of different types, just like a structure.
- ➔ However, with unions, you can only store information in one field at any one time.
- ➔ You can picture a union as like a chunk of memory that is used to store variables of different types.
- ➔ Once a new value is assigned to a field, the existing data is wiped over with the new data.
- ➔ A union can also be viewed as a variable type that can contain many different variables (like a structure), but only actually holds one of them at a time (not like a structure).
- ➔ This can save memory if you have a group of data where only one of the types is used at a time.
- ➔ The size of a union is equal to the size of its largest data member.
- ➔ In other words, the C compiler allocates just enough space for the largest member.
- ➔ This is because only one member can be used at a time, so the size of the largest, is the most you will need.

- ➔ Here is an example:

```
...
union time
{
        long simpleDate;
        double perciseDate;
}mytime;
```

- ➔ To access the fields of a union, use the dot operator(.) just as you would for a structure.
- ➔ When a value is assigned to one member, the other member(s) get whipped out since they share the same memory.
- ➔ Using the example above, the precise time can be accessed like this:

```
. . .
printTime( mytime.perciseDate );
. . .
```

- ➔ Here is a sample program to illustrate the use of unions.

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

```
void main()
{
	union data
	{
		char a;
		int x;
		float f;
	} myData;

	int mode = 1;
	myData.a = 'A';
	printf("Here is the Data:\n%c\n%i\n%.3f\n", myData.a, myData.x, myData.f );

	myData.x = 42;
	mode = 2;
	printf("Here is the Data:\n%c\n%i\n%.3f\n", myData.a, myData.x, myData.f );

	myData.f = 101.357;
	mode = 3;
	printf("Here is the Data:\n%c\n%i\n%.3f\n", myData.a, myData.x, myData.f );

	if( mode == 1 )
	  printf("The char is being used\n");
	else if( mode == 2 )
	  printf("The int is being used\n");
	else if( mode == 3 )
	  printf("The float is being used\n");
	getch();
}
```

This little program declares a union with an int, float, and char. It uses each field, and after each use prints out all the fields (with one ugly printf statement). Here is some sample output:

```
Here is the Data:
A
577
0.000
Here is the Data:
*
42
```

0.000
Here is the Data:

1120581321
101.357
The float is being used

➔ Union allows same storage to be referenced in different ways.
➔ Only one way is valid at any given time

➢ Usage

- ✓ access individual bytes of larger type
- ✓ variable format input records (coded records)
- ✓ sharing an area to save storage usage
- ✓ unions not used nearly as much as structures
- ✓ sizeof union is size of its biggest member
- ✓ Unions most often contain different types of structures
- ✓ Can only initialize first member of union
- ✓ Can assign (copy) one union variable to another
- ✓ Can pass union or pointer to union as function arg
- ✓ Function can return union type
- ✓ Can define pointers to union type object
- ✓ Members accessed as unionvar.member or unionptr-$gt;member
- ✓ Syntax, format and use of tags and declarators like struct, but members overlay each other, rather than following each other in memory.

Ch-8

File Handling in C

- ✓ Concept of Data Files
- ✓ File Handling in C
- ✓ Opening a File
- ✓ Reading from a File
- ✓ Closing a File
- ✓ Functions: fgets() and fputs()
- ✓ Functions: fprintf() and fscanf()
- ✓ Random access using fseek()
- ✓ ftell()
- ✓ rewind()
- ✓ feof()
- ✓ ferror()
- ✓ Functions: fwrite() and fread()
- ✓ File I/O (using TEXT File)
- ✓ Command Line Arguments

❖ Concept of Data Files

- ➔ If you work with a computer you work with files. Sometimes those files contain information about something you are writing - in a word processor, for example.
- ➔ Sometimes those files contain other information like results of calculations.
- ➔ Sometimes those files contain results of measurements you took in a laboratory or out in the field.
- ➔ In this lesson we are going to examine data files.
- ➔ There are numerous good reasons why you need to understand some basic ideas about data files.
    - The amount of data you can store in a data file on a disk is determined by the precision of the measurements you take and the number of data points you want to record.
    - You may want to write programs that take and store data, and you need to understand a little bit about file structure when you do that.
- ➔ In both cases you need to know something about file structure and some details of the characteristics of data files.
- ➔ That's what this lesson is about.

❖ File handling in C

- ➔ A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind.
- ➔ The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image.
- ➔ There are two kinds of files that programmers deal with text files and binary files.
- ➔ C communicates with files using a new datatype called a file pointer.
- ➔ This type is defined within stdio.h, and written as FILE *.
- ➔ A file pointer called output_file is declared in a statement like

```
FILE *output_file;
```

❖ Opening a File

- ➔ **fopen** is used to open a file for read, write or update.
- ➔ The first statement declares the variable fp as a pointer to the data type FILE.
- ➔ As stated earlier, File is a structure that is defined in the I/O Library.
- ➔ The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.
- ➔ fopen() contain the file name and mode (the purpose of opening the file).
    - r is used to open the file for read only.
    - w is used to open the file for writing only.

- a is used to open the file for appending data to it.

```
Library:   stdio.h
Prototype: FILE *fopen(const char *filename, const char *mode);

Syntax:    FILE *fp;
             fp = fopen( "/etc/printcap", "r");
```

➔ Example…

```
#include <stdio.h>
#include <conio.h>
void main ()
{
    FILE *fp;
    fp = fopen("data.txt", "r")
    if (fp == NULL)
    {
       printf("\n\n File does not exist, please check!\n");
    }
     fclose(fp);
}
```

❖ Reading from a File

➔ File reading is done by fopen().

➔ Contents are brought into buffer and a pointer is set up that point to the first character in to the buffer.

➔ To read the file's contents from the memory, there is a function used named getc().

❖ Closing a File

➔ A file must be closed as soon as all operations on it have been completed.

➔ This would close the file associated with the file pointer.

➔ The input output library supports the function to close a file.

```
Library:   stdio.h

Prototype: int fclose( FILE *stream);

Syntax:    FILE *fp;
           fclose(fp);
```

➔ Example

```
#include <stdio.h>
#include <conio.h>
void main()
 {
     FILE *myfile;
     char c;
     myfile = fopen("firstfile.txt", "r");
     if (myfile == NULL)
     {
         printf("File doesn't exist\n");
     }
     else
     {
         do
         {
                 c = getc(myfile);
                 putchar(c);
          } while (c != EOF);
     }
     fclose(myfile);
     getch();
 }
```

❖ Functions: fgets() and fputs()

- ➔ These are useful for reading and writing entire lines of data to/from a file.
- ➔ If *buffer* is a pointer to a character array and *n* is the maximum number of characters to be stored, then…
  - ✓ *fgets (buffer, n, input_file);*
- ➔ will read an entire line of text (max chars = n) into *buffer* until the newline character or n=max, whichever occurs first.
- ➔ The function places a NULL character after the last character in the buffer.
- ➔ The function will be equal to a NULL if no more data exists.
  - ✓ *fputs (buffer, output_file);*
- ➔ It writes the characters in *buffer* until a NULL is found. The NULL character is not written to the *output_file*.

- ➔ Example…

```
#include <stdio.h>
#include <conio.h>
#define MAXLINE 20
void main()
```

```
{
   char line[MAXLINE];
   while (fgets(line, MAXLINE, stdin) != NULL && line[0] != '\n')
   {
            fputs(line, stdout);
   }
   getch();
}
```

❖ Functions: fprintf() and fscanf()

- ➔ Once a file has been successfully opened, you can read from it using fscanf() or write to it using fprintf().
- ➔ These functions work just like scanf() and printf(), except they require an extra first parameter, aFILE * for the file to be read/written.
- ➔ The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files.
- ➔ The first argument of these functions is a file pointer which specifies the file to be used.
- ➔ The general form of fprintf is

  fprintf(fp,"control string", list);

  where fp id a file pointer associated with a file that has been opened for writing.

- ➔ The control string is file output specifications list may include variable, constant and string.

  fprintf(f1,%s%d%f",name,age,7.5);

  Here, name is an array variable of type char and age is an int variable

- ➔ The general format of fscanf is

  fscanf(fp,"controlstring",list);

  This statement would cause the reading of items in the control string.

  fscanf(f2,"5s%d",item,&quantity");
- ➔ Like scanf, fscanf also returns the number of items that are successfully read.
- ➔ Example 1

```
/*Program to handle mixed data types*/
#include< stdio.h >
void main()
{
FILE *fp;
int num,qty,I;
float price,value;
```

```
    char item[10],filename[10];

    printf("Input filename");
    scanf("%s",filename);

    fp=fopen(filename,"w");
    printf("Input inventory datann"0;
    printf("Item namem number price quantityn");

    for I=1;I< =3;I++)
    {
        fscanf(stdin,"%s%d%f%d",item,&number,&price,&quality);
        fprintf(fp,"%s%d%f%d",itemnumber,price,quality);
     }
     fclose (fp);
     fprintf(stdout,"nn");
     fp=fopen(filename,"r");

    printf("Item name number price quantity value");
    for(I=1;I< =3;I++)
    {
        fscanf(fp,"%s%d%f%d",item,&number,&prince,&quality);
        value=price*quantity");
        fprintf("stdout,"%s%d%f%d%dn",item,number,price,quantity,value);
    }
    fclose(fp);
    getch();
}
```

➔ Example 2

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 40

void main()
{

   FILE *fp;
   char words[MAX];
   if ((fp = fopen("wordy", "a+")) == NULL)
   {
```

```
        fprintf(stdout,"Can't open \"words\" file.\n");
        exit(1);
    }
    puts("Enter words to add to the file; press the Enter");
    puts("key at the beginning of a line to terminate.");
    while (gets(words) != NULL  && words[0] != '\0')

        fprintf(fp, "%s ", words);

    puts("File contents:");
    rewind(fp);          /* go back to beginning of file */
    while (fscanf(fp,"%s",words) == 1)
        puts(words);

    if (fclose(fp) != 0)
        fprintf(stderr,"Error closing file\n");

    getch();

}
```

❖ Random access using fseek()

- ➔ Sometimes it is required to access only a particular part of the and not the complete file.
- ➔ The general format of fseek function is a s follows:
  fseek(file pointer, offset, position);
- ➔ This function is used to move the file position to a desired location within the file.
- ➔ Fileptr is a pointer to the file concerned.
- ➔ Offset is a number or variable of type long, and position in an integer number.
- ➔ Offset specifies the number of positions (bytes) to be moved from the location specified but the position.
- ➔ The position can take the 3 values.
- ➔ Value Meaning
  0 Beginning of the file
  1 Current position
  2 End of the file.
- ➔ SEEK_SET : seek 'offset' number of bytes from the beginning
- ➔ SEEK_CURR : seek 'offset' number of bytes from the current position
- ➔ SEEK_END : seek 'offset' number of bytes from the end of the file.
- ➔ Example…

```
#include <stdio.h>
#include <conio.h>
void main ()
{
        FILE * f;
        f = fopen ( "myfile.txt" , "w" );
        fputs ( "Hello World" , f );
        fseek ( f , 6 , SEEK_SET );
        fputs ( " India" , f );
        fclose ( f );
        return 0;
}
```

❖ ftell()

- ➔ The function *ftell* returns the current offset in a stream in relation to the first byte.
- ➔ Returns the current value of the position indicator of the *stream*.
- ➔ For binary streams, the value returned corresponds to the number of bytes from the beginning of the file.
- ➔ For text streams, the value is not guaranteed to be the exact number of bytes from the beginning of the file, but the value returned can still be used to restore the position indicator to this position using fseek.
- ➔ Example

```
/* ftell example : getting size of a file */
#include <stdio.h>
#include <conio.h>
void main ()
{
    FILE * pFile;
    long size;
     pFile = fopen ("myfile.txt","rb");
     if (pFile==NULL)
            perror ("Error opening file");
     else
     {
        fseek (pFile, 0, SEEK_END);
        size=ftell (pFile);
        fclose (pFile);
        printf ("Size of myfile.txt: %ld bytes.\n",size);
      }
      getch();
}
```

❖ rewind()

➔ **Set position indicator to the beginning.**

```
void rewind ( FILE * stream );
```

➔ Sets the position indicator associated with *stream* to the beginning of the file.
A call to rewind is equivalent to:

```
fseek ( stream , 0L , SEEK_SET );
```

➔ Example

```
/* rewind example */
#include <stdio.h>
#include <conio.h>
void main ()
{
        int n;
        FILE * pFile;
        char buffer [27];

        pFile = fopen ("myfile.txt","w+");
        for ( n='A' ; n<='Z' ; n++)
        {
          fputc ( n, pFile);
        }
        rewind (pFile);
        fread (buffer,1,26,pFile);
        fclose (pFile);
        buffer[26]='\0';
        puts (buffer);
        getch();
}
```

❖ feof()

➔ Checks whether the End-of-File indicator associated with *stream* is set, returning a value different from zero if it is.

➔ This indicator is generally set by a previous operation on the *stream* that reached the End-of-File.

```
int feof ( FILE * stream );
```

➔ Further operations on the stream once the End-of-File has been reached will fail until either rewind, fseek or fsetpos is successfully called to set the position indicator to a new value.

➔ Example…

```
/* feof example: byte counter */
#include <stdio.h>
#include <conio.h>
void main ()
{
  FILE * pFile;
  long n = 0;
  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    while (!feof(pFile))
    {
      fgetc (pFile);
      n++;
      }
    fclose (pFile);
    printf ("Total number of bytes: %d\n", n-1);
  }
  getch();
}
```

❖ ferror()

➔ Checks if the error indicator associated with *stream* is set, returning a value different from zero if it is.

➔ This indicator is generaly set by a previous operation on the *stream* that failed.

```
int ferror ( FILE * stream );
```

➔ Example

```
/* ferror example: writing error */
#include <stdio.h>
#include <conio.h>
void main ()
{
  FILE * pFile;
  pFile=fopen("myfile.txt","r");
  if (pFile==NULL)
     perror ("Error opening file");
  else
  {
    fputc ('x',pFile);
```

```
    if (ferror (pFile))
     {
            printf ("Error Writing to myfile.txt\n");
     }
    fclose (pFile);
  }
  getch();
}
```

❖ Functions: fread() and fwrite()

➢ fread()

➔ Reads an array of *count* elements, each one with a size of *size* bytes, from the *stream* and stores them in the block of memory specified by *ptr*.

➔ The postion indicator of the stream is advanced by the total amount of bytes read.

➔ The total amount of bytes read if successful is (*size* * *count*).

➔ Syntax:

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

➔ Parameters

- ptr
  - Pointer to a block of memory with a minimum size of (*size***count*) bytes.
- size
  - Size in bytes of each element to be read.
- count
  - Number of elements, each one with a size of *size* bytes.
- stream
  - Pointer to a FILE object that specifies an input stream.

➔ Example

```
/* fread example: read a complete file */
#include <stdio.h>
#include <stdlib.h>

void main ()
{
  FILE * pFile;
  long lSize;
  char * buffer;
  size_t result;

  pFile = fopen ( "myfile.bin" , "rb" );
```

```
  if (pFile==NULL) {fputs ("File error",stderr); exit (1);}

  // obtain file size:
  fseek (pFile , 0 , SEEK_END);
  lSize = ftell (pFile);
  rewind (pFile);

  // allocate memory to contain the whole file:
  buffer = (char*) malloc (sizeof(char)*lSize);
  if (buffer == NULL) {fputs ("Memory error",stderr); exit (2);}

  // copy the file into the buffer:
  result = fread (buffer,1,lSize,pFile);
  if (result != lSize) {fputs ("Reading error",stderr); exit (3);}

  /* the whole file is now loaded in the memory buffer. */

  // terminate
  fclose (pFile);
  free (buffer);
  getch();
}
```

➢ fwrite()

➔ Writes an array of *count* elements, each one with a size of *size* bytes, from the block of memory pointed by *ptr* to the current position in the *stream*.

➔ The postion indicator of the stream is advanced by the total number of bytes written. The total amount of bytes written is (size * count).

➔ Syntax

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

➔ Parameters

- ptr
  - Pointer to the array of elements to be written.
- size
  - Size in bytes of each element to be written.
- count
  - Number of elements, each one with a size of *size* bytes.
- stream
  - Pointer to a FILE object that specifies an output stream.

➔ Example

```
 /* fwrite example : write buffer */
#include <stdio.h>
#include <conio.h>
void main ()
{
  FILE * pFile;
  char buffer[ ] = { 'x' , 'y' , 'z' };
  pFile = fopen ( "myfile.bin" , "wb" );
  fwrite (buffer , 1 , sizeof(buffer) , pFile );
  fclose (pFile);
  getch();
}
```

❖ File I/O (Using TEXT File)

➔ I/O function library is:

Function Name	Task
fopen	Opens a text file.
fclose	Closes a text file.
feof	Detects end-of-file marker in a file.
fscanf	Reads formatted input from a file.
fprintf	Prints formatted output to a file.
fgets	Reads a string from a file.
fputs	Prints a string to a file.
fgetc	Reads a character from a file.
fputc	Prints a character from a file.

❖ Command Line Arguments

➔ In C it is possible to accept command line arguments.

➔ Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

➔ To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments.

➔ In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

➔ The full declaration of main looks like this:

```
int main ( int argc, char *argv[ ] )
```

- ➔ The integer, argc is the **arg**ument **c**ount.
- ➔ It is the number of arguments passed into the program from the command line, including the name of the program.
- ➔ The array of character pointers is the listing of all the arguments.
- ➔ argv[0] is the name of the program, or an empty string if the name is not available.
- ➔ After that, every element number less than argc is a command line argument.
- ➔ You can use each argv element just like a string, or use argv as a two dimensional array.
- ➔ argv[argc] is a null pointer.
- ➔ How could this be used? Almost any program that wants its parameters to be set when it is executed would use this.
- ➔ One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen.
- ➔ Example…

```
#include <stdio.h>
#include <conio.h>
void main ( int argc, char *argv[] )
{
   if ( argc != 2 ) /* argc should be 2 for correct execution */
   {
      /* We print argv[0] assuming it is the program name */
      printf( "usage: %s filename", argv[0] );
   }
   else
   {
      // We assume argv[1] is a filename to open
      FILE *file = fopen( argv[1], "r" );

      /* fopen returns 0, the NULL pointer, on failure */
      if ( file == 0 )
      {
         printf( "Could not open file\n" );
      }
      else
      {
         int x;
```

```
            while  ( ( x = fgetc( file ) ) != EOF )
            {
               printf( "%c", x );
            }
            fclose( file );
         }
      }
      getch();
   }
```

Ch-9 Misc

- ✓ Type Casting
- ✓ Typedef
- ✓ Symbolic Constants
- ✓ C Preprocessors

❖ Type Casting

- ➔ Previously, you learned that the value of a variable is stored as a sequence of bits, and the data type of the variable tells the compiler how to translate those bits into meaningful values.
- ➔ Often it is the case that data needs to be converted from one type to another type.
- ➔ This is called **type casting.**
- ➔ It refers to different ways of, implicitly or explicitly, changing an entity of one data type into another.
- ➔ This is done to take advantage of certain features of type hierarchies or type representations.
- ➔ There are two methods:
    - Implicit
    - Explicit

1. Implicit Conversion

- ➔ **Implicit type conversion** is done automatically by the compiler whenever data from different types is intermixed.
- ➔ When a value from one type is assigned to another type, the compiler implicitly converts the value into a value of the new type.
- ➔ Example…
  short int → int → unsigned int → long int → unsigned long int → float → double → long double.

2. Explicit Conversion

- ➔ It has higher priority than Implicit Conversion.
- ➔ Syntax:
  (data_type) operand
    - ✓ Operand can be variable or phrase.

❖ Typedef

- ➔ C provides a feature called "TYPEDEF" that allows user to define an identifier that would represent an existing data type.
- ➔ Typedef is used to redefine the name of an existing variable type.
- ➔ Syntax:
  typedef data_type variable;
- ➔ Advantage:
  We can create meaningful data type names for increasing the readability of the program.

➔ Example:

```
typedef int dig;
typedef float avg;
```

dig symbolizes int & avg symbolizes float data type.

Now, we can write that…

```
dig a,b;
avg m;
```

❖ Symbolic Constants

➔ Constants defined in the header file are called symbolic constants.
➔ They are also known as MACRO PRE-PROCESSOR.
➔ Constants values are assigned to the variable at beginning of the program.
➔ It can be defined using pre-processor directive #define.
➔ Naming conventions are same as variable names.
➔ Value of Symbolic constants can't be changed during execution of program.
➔ They are written in CAPS form, just to make difference from other variables.
➔ Features are:
1. Modifiability
2. Understandability

➔ Example:

```
#define PI 3.14
```

❖ C Preprocessors

➔ It is a program that processes the source code before it passes through the compiler.
➔ A preprocessor directive begins with a # symbol. Preprocessor directives may be placed anywhere in the program but are generally placed in the beginning of the program.
➔ The different preprocessor directives are:
1. Macro expansion (#define)
2. File Inclusion (#include)
3. Conditional compilation (#ifdef)

1.) Macro Expansion

✓ E.g. #define MAX 25
✓ In the above example MAX is called the Macro template and 25 is called the macro expansion.
✓ Wherever the macro template is present in the program the preprocessor

replaces it with the expansion.

- ✓ Macro expansion makes the program easier to read and modify.
- ✓ Since the template MAX is used in the program it indicates some maximum value.
- ✓ Also any modifications to the value can be made at the definition. Hence the entire program need not be changed.
- ✓ **Macro with arguments:**
  - #define AREA(x) (3.14*x*x)

2.) File Inclusion

- ✓ This directive causes the filename to be included in the program.
- ✓ By this all the contents of the specified file are placed at the point of directive.
- ✓ A large file may be divided into several different files each containing a set of related functions.
- ✓ These files are then included wherever the functions are required.
- ✓ Also some functions may be required in most of the program.
- ✓ In such cases, the commonly needed function can be stored in a file and that file can be #included in the program.

  E.g. #include <filename> or #include "filename"

- ✓ <u>#include can be written in two ways:</u>
  - #include<filename> :
    - It causes the filename to be searched in the specified library path only.
  - #include "filename" :
    - It causes the filename to be searched in the current directory as well as the specified library path.

3.) Conditional Compilation

- ✓ This causes only a part of the source code to be complied and converted into executable code.
- ✓ Syntax:

```
#ifdef macro              // if the macro is defined
#else
#endif
#ifndef macro             //if the macro is not defined
#if  condition            // if the condition is true
#elif
```

- ✓ Example:

```
#define MAX 25
void main()
{
                                        #ifdef MAX
                                        printf("……");
```

```
    #endif
    #if TEST>10
    #endif
}
```