UNIT – I

INTRODUCTION

Computer has emerged as the most useful machine in recent times. It can perform wide variety of tasks like receiving data, processing it, and producing useful results. However, being a machine, the computer cannot perform on its own. A computer needs to be instructed to perform even a simple task like adding two numbers. Computers work on a set of instructions called computer program, which clearly specify the ways to carry out a task. A computer takes instructions, in the form of computer programs, and carry out the requested task.

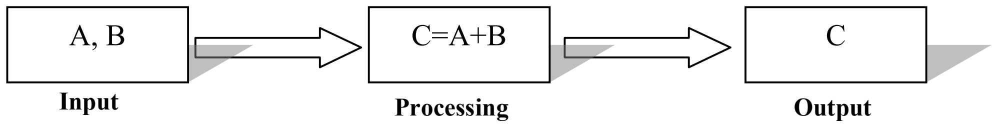
The human beings, use natural languages such as English, Spanish, or French to communicate. Similarly, a user communicates with the computer in a language understood by it. The instructions, provided in the form of computer programs, are developed using computer or programming languages.

DEVELOPING A PROGRAM

A program consists of a series of instructions that a computer processes to perform, the required operation. In addition, it also contains some fixed data, required to perform the instructions, and the process of defining those instructions and data. Thus, in order to design a program, a programmer must determine three basic rudiments:

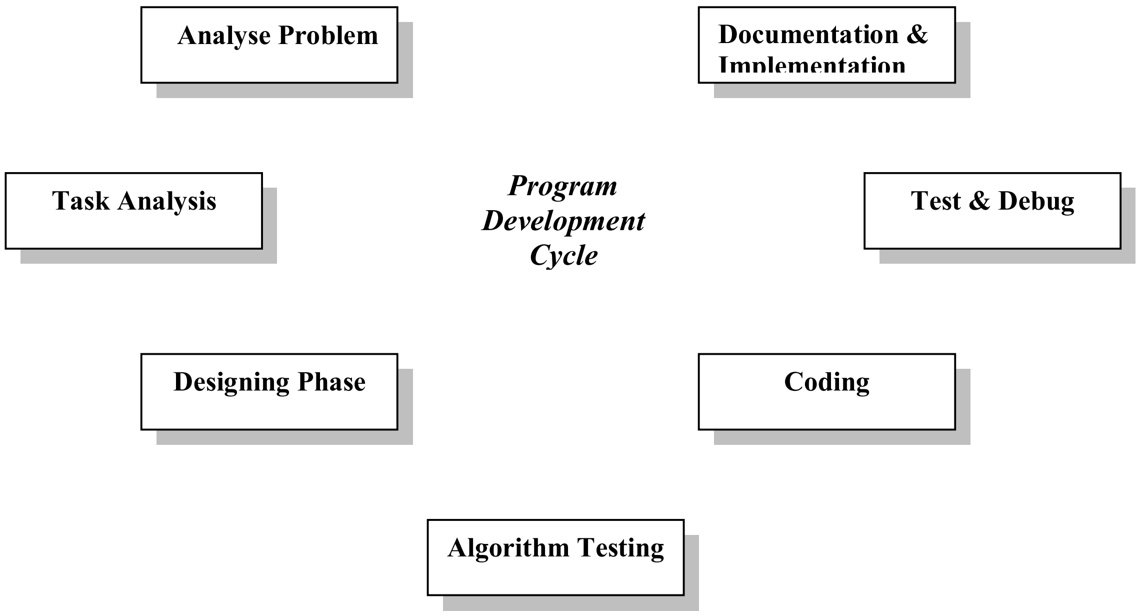
1. The instructions to be performed.
2. The order in which those instructions are to be performed.
3. The data required to perform those instructions.

To perform a task using a program, a programmer has to consider various inputs of the program along with the process, which is required to convert input into desired output. Suppose we want to calculate the sum of two numbers, A and B, and store the sum in C, here A and B are the inputs, addition is the process, and C is the output of the program.



PROGRAM DEVELOPMENT CYCLE

Before starting the process of writing a program (coding), the programmer has to determine the problem that needs to be solved. There are different approaches to problem solving. Most require breaking the problem into a series of smaller steps, independent of the programming language. One common technique is to use the program development cycle, with the number of steps that may vary according to the person who has formalized the development. Development cycle of a program includes the following phases.



1. Analyse/Define the Problem: The problem is analyzed precisely and completely. Based on understanding, the developer knows about the scope within which the problem needs to be analysed.

2. Task Analysis: After analysing the problem, the developer needs to develop various solutions to solve the given problem. From these solutions, the optimum solution is chosen, which can solve the problem comfortably and economically.

3. Designing Phase: After selecting the appropriate solution, algorithm is developed to depict the basic logic of the selected solution. An algorithm depicts the solution in logical steps. Further, algorithm is represented by flowcharts and pseudocodes. These tools make program logic clear and they eventually help for coding. This phase is also known as design phase.

4. Testing the Algorithm for Accuracy: Before converting the algorithms into actual code, it should be checked for accuracy. The main purpose of checking algorithm is to identify major logical errors at an early stage, because logical errors are often difficult to detect and correct at later stages.

5. Coding: After meeting all the design considerations, the actual coding of the program takes place in the chosen programming language. Depending upon application domain and available resources, a program can be written by using computer languages of different levels such as machine, assembly or high-level languages.

6. Test and Debug the Program: It is common for the initial program code to contain errors, a program compiler and programmer-designed test data machine tests the code for syntax errors. The results obtained are compared with results calculated manually from this test data. Depending upon the complexity of the program, several rounds of testing may be required.

7. Documentation and Implementation: Once the program is free from all the errors, it is the duty of the program developers to ensure that the program is supported by suitable documentation. After documentation, the program is installed on the end user's machine.

PROBLEM SOLVING TECHNIQUES

Problem solving technique is a set of techniques and graphical tools that helps in providing in logic for solving a problem. In other words, these tools are used to express the logic of the problem by specifying the correct sequence of all instructions to be carried out. Following sections; in discuss the various problem-solving tools, which are algorithms, flowcharts, and pseudocodes.

ALGORITHMS

Algorithms are one of the most basic tools that are used to develop the problem solving logic. An algorithm is defined as a finite sequence of explicit instructions that, when provided with a set of input values produces an output and then terminates. In algorithm, after a finite number of steps, solution of the problem is achieved. Algorithms can have steps that repeat (iterate) or require decisions (logic and comparison) until the task is completed.

Different algorithms may accomplish the same task, with a different set of instructions, in more or less the same time, space, and efforts. For example, two different recipes for preparing tea, one ' add the sugar' while 'boiling the water' and the other 'after boiling the water' produce the same result. However, performing an algorithm correctly does not guarantee a solution, if the algorithm is flawed or not appropriate to the context. For example, preparing the tea algorithm will fail if there were no tea leaves present; even if all the motions of preparing the tea were performed as if the tea leaves were there. We use algorithms in our daily life.

For example, to determine the largest number out of three numbers A, B, and C, the following algorithm may be used.

```
Step 1: Start
Step 2: Read three numbers say A, B, C
Step 3: Find the larger number between A and B and store it in MAX_AB
```

Step 4: Find the larger number between MAX_AB and C and store it in MAX

Step 5: Display MAX

Step 6: Stop

The above-mentioned algorithm terminates after six steps. This explains the feature of finiteness. Every action of the algorithm is precisely defined; hence, there is no scope for ambiguity. Once the solution is properly designed, the only job left is to code that logic into a programming language.

Algorithm Properties

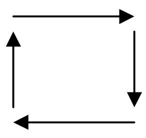
The properties of Algorithm are:

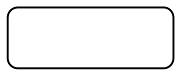
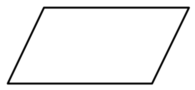
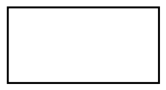
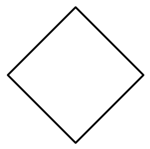
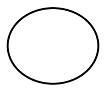
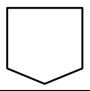
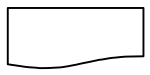
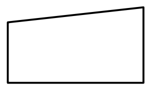
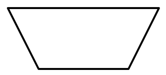
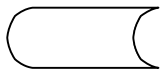
1. There must be no ambiguity in any instruction.
2. There should not be any uncertainty about which instruction is to be executed next.
3. The algorithm should conclude after a finite number of steps. An algorithm cannot be open-ended.
4. The algorithm must be general enough to deal with any contingency.

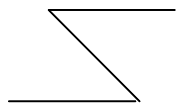
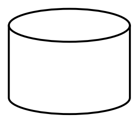
FLOWCHART

A flowchart is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. The boxes represent operations and the arrows represent the sequence in which the operations are implemented. The primary purpose of the flowchart is to help the programmer in understanding the logic of the program.

Flowcharts can be compared with the blueprint of a building. Just as an architect draws a blueprint before starting the construction of a building, a programmer draws a flowchart before writing a computer program. As in the case of the drawing of a blueprint, the flowchart is drawn according to defined rules and using standard flowchart symbols prescribed by ,American National Standard Institute (ANSI). Some standard symbols that are frequently required for flowcharts are shown below.

Symbol	Symbol Name	Description
	Flow Lines	Flow lines are used to connect symbols. These lines indicate the sequence of steps and the direction of flow of control.

	Terminal	This symbol is used to represent the beginning (start), the termination (end), or halt (pause) in the program logic.
	Input/Output	It represents information entering or leaving the system, such as customer order (input) and servicing (output).
	Processing	Process symbol is used for representing arithmetic and data movement instructions. It can represent a single step ('add two cups of flour'), or an entire sub-process ('make bread') within a larger process.
	Decision	Decision symbol denotes a decision (or branch) to be made. The program should continue along one of the two routes (IF/ELSE). This symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is yes or no.
	connector	Connector symbol is used to join different flow lines.
	Off-page Connector	This symbol is used to indicate that the flowchart continues on the next page.
	Document	Document is used to represent a paper document produced during the flowchart process.
	Manual Input	Manual input symbol represents input to be given by a developer / programmer .
	Manual Operation	Manual operation symbol shows that the process has to be done by a developer/programmer.
	Online Storage	This symbol represents the online data storage such as hard disks, magnetic drums, or other storage devices.

	Communication Link	Communication link symbol is used to represent data received or to be transmitted from an external system
	Magnetic Disk	This symbol is used to represent data input or output from and to a magnetic disk.

Guidelines for Preparing flowcharts The following guidelines should be used for creating a flowchart:

- The flowchart should be clear, neat, and easy to follow.
- The flowchart must have a logical start and finish.
- In drawing a proper flowchart, all necessary requirements should be listed in logical order.
- Only one flow line should come out from a process symbol.
- Only one flow line should enter a decision symbol. However, two or three flow lines (one for each possible answer) may leave the decision symbol.
- Only one flow line is used with a terminal symbol.
- In case of complex flowcharts, connector symbols are used to reduce the number of flow lines.

Benefits of-Flowcharts A flowchart helps to clarify how things are currently working and how they could-be improved. The reasons for using flowcharts as a problem-solving tool are given below.

- **Makes Logic Clear**: The 'main advantage of using a flowchart to plan a task is that it provides a pictorial representation of the task, which makes the logic easier to follow. Even less experienced personnel can trace the actions represented by a flowchart, that is, flowcharts are ideal for visualizing fundamental control structures employed in computer programming.
- **Communication:** Being a graphical representation of a problem-solving logic, flowcharts are better way of communicating the logic of a system to all concerned. The diagrammatical representation of logic is easier to communicate to all the interested parties as compared to actual program cede as the users may not be aware of all the programming techniques and jargons.

- **Effective Analysis:** With the help of a flowchart, the problem can be analysed in an effective way. This is because the analysing duties of the programmers can be delegated to other persons, who may or may not know the programming techniques, but they have a broad idea about the logic.
- **Useful in Coding**: The flowcharts act as a guide or blueprint during the analysis and program development phase. Once the flowcharts are ready, the programmers can plan the coding process effectively as they know where to begin and where to end, making sure that no steps are omitted. As a result, error free programs are developed in high-level languag and that too at a faster rate.
- **Proper Testing and Debugging**: By nature, a flowchart helps in detecting the errors in a program, as the developers know exactly what the logic should do.
- **Appropriate Documentation:** Flowcharts serve as a good program documentation tool. Since normally the programs are developed for novice users; they can take the help of the program documentation to know what the program actually does and how to, use the program.

Limitations of Flowcharts Flowchart can be used for designing the basic concept of the program in pictorial form but cannot be used for programming purposes. Some of the limitations of the flowchart are given as follows:

- **Complex:** The major disadvantage in using flowcharts is that when a program is very large, the flowcharts may continue for many pages, making them hard to follow.
- **Costly:** Drawing flowcharts are viable only if the problem-solving logic is straightforward and not very lengthy. However, if flowcharts are to be drawn for a huge application, the time and cost factor of program development may get out of proportion, making it a costly affair.
- **Difficult to Modify:** Due to its symbolic nature, any changes or modification to a flowchart usually requires redrawing the entire logic again, and redrawing a complex flowchart is not a simple task. It is not easy to draw thousands of flow lines and symbols along with proper spacing, especially for a large complex program.
- **No Update:** Usually programs are updated regularly. However, the corresponding update of flowcharts may not take place, especially in the case of large programs. As a result, the logic used in the flowchart may not match with the actual program's logic.

PSEUDOCODE

Pseudocode (pronounced soo-doh-kohd) is made up of two words: pseudo and code. Pseudo means imitation and code refers to instructions, written in a programming language. As the name suggests, pseudocode is not a real programming code, but it models and may even look like programming code. Pseudocode uses plain English statements rather than symbols, to represent the processes of a computer program. It is also known as PDL (Program Design Language), as it emphasises more on the design aspect of a computer program or structured English, because usually pseudocode instructions are written in normal English, but in a structured way.

If an algorithm is written in English, the description may be at such a high level that it may prove difficult to analyse the algorithm and then to transform it into actual code. If instead, the algorithm is written in code, the programmer has to invest a lot of time in determining the details of an algorithm, which he may choose not to implement (since, typically, algorithms are analysed before deciding which one to implement). Therefore, the goal of writing pseudocode is to provide a high-level description of an algorithm, which facilitates analysis and eventual coding, but at the same time suppresses many of the details that are insignificant. For example, the pseudocode given below calculates the area of a rectangle.

```
PROMPT the user to enter the height of the rectangle
PROMPT the user to enter the width bf the rectangle
COMPUTE the area by multiplying the height with width
DIS'PLAY the area
STOP
```

Pseudocode uses some keywords to denote programming processes. Some of them are:

- **Input:** READ, OBTAIN, GET, and PROMPT
- **Output:** PRINT, DISPLAY, and SHOW
- **Compute:** COMPUTE, CALCULATE, and DETERMINE
- **Initialize:** SET and INITIALISE
- **Add One:** INCREMENT

Pseudocode Guidelines Writing pseudocode is not a difficult task. Even if you do not know anything about the computers or computer languages, you can still develop effective and efficient pseudo codes, if you are writing in an organised manner.

Here are a few general guidelines for developing pseudocodes:

- Statements should be written in simple English and should be programming language independent. Remember that pseudocodes only describe the logic plan to develop a program, it is not programming.
- Steps must be understandable, and when the steps are followed, they must produce a solution to the specified problem.
- Pseudocodes should be concise.
- Each instruction should be written in a separate line and each statement in pseudocode should express just one action for the computer.
- Capitalize keywords such as READ, PRINT, and so on.
- Each set of instructions is written from top to bottom, with only one entry and one exit.
- It should allow for easy transition from design to coding in programming language.

Benefits of Pseudocode Pseudocode provides a simple method of developing the program logic as it uses everyday language to prepare a brief set of instructions in the order in which they appear in the completed program. It allows the programmer to focus on the steps required to solve a program rather than on how to use the computer language. Some of the most significant benefits of pseudocode are:

- Since it is language independent, it can be used by most programmers.
- It is easier to develop a program from a pseudocode than with a flowchart.
- Often, it is easy to translate pseudocode into a programming language, a step which can be accomplished by less experienced programmers.
- Unlike flowcharts, pseudocode is compact and does not tend to run over many pages. Its simple structure and readability makes it easier to modify.

Limitations of Pseudocode Although pseudocode is a very simple mechanism to simplify problem-solving logic, it has its limitations. Some of the most notable limitations are:

- It does not provide visual representation of the program's logic.
- There are no accepted standards for writing pseudocodes.
- Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules.

HISTORY OF C

C is a high level programming language. C language is derived from two languages BCPL & B in 1960 at Cambridge University. C language is developed at Bell Laboratories in 1972. Dennis Ritchie invented C language. C Language mainly evolved from ALGOL, BCPL and B. C is developed for programming in OS called UNIX.

FEATURES OF "C" LANGUAGE

- C is a Structured Programming Language and powerful language.
- C is a versatile Language (i.e.,) it has features of both Low Level Languages and High Level Languages.
- It is used in system programming. Compilers, packages, Operating System are written using C.
- C is a highly portable language. Source Program written in one computer can be compiled, executed and run on another computer with less modification.
- Users can develop program quickly with few mistakes.
- It increases the readability of Program. Program can be modified without affecting the other parts of a program.
- C is a Case Sensitive Language. It differentiates both Upper Case and Lower Case separately.
- Every statement in C ends with semicolon.
- Programmers can write additional functions of their own. C language has a C Compiler.
- C language has the ability to extend itself. C is also a free form language, because each group of statements can be placed together on one line.
- C language is used for developing software packages, system software, database systems, graphics packages, spread sheets, CAD/CAM applications, word processors, office automation, scientific and engineering applications.

STRUCTURE OF C PROGRAM

```
Documentation Section
Header File Section / Link Section
Definition Section
Global (Variable) Declaration Section
main()
{
        Declaration(local variable) part
        Executable part
}
sub program section
function 1
function 2
‘
‘
Function n
```

Documentation Section

Documentation section consists of set of command lines. It is included in the first line of a C program. It consists of comments. There are two types of comments. They are Single line Comments and Multi line comments. Comments are used to give a meaningful name to a C Program.

Single line comments are specified with two backslash // or /*’ and ended with */ in a same Line. Multi line comments are specified by a backslash (/) followed by an asterisk (*) symbol and ended with asterisk symbol (*) followed by Backslash (/).

Link section/Header File section

In this section we can declare the header files of the program. Header file consists of predefined function definitions for frequently used functions. Some of the functions in C programs was already and written, executed and stored in the header files. Whenever we need to use the particular function, we should simply mention the name in the link section. Link section provides instructions to compiler to link functions from system library.

Definition Section

It is used to define symbolic constants.

Global declaration section

Used to declare functions, variables or symbolic constants globally. Two types of declaration are allowed in C Language.

main( ) section

The execution of C program starts from the main function. Every C program must have a main function. It contains two parts. They are declaration part and executable part. Declaration part and executable part must appear between opening **‘ { ‘** and closing braces ‘ **}**’. Opening brace specifies the beginning of C program and close brace specifies the logical end of C program.

Declaration part

Declaration part is used to declare variables. Every variable must be declared before using it in C program. Every variable are specified with their data type followed by a name. If there are more variables of same data type then they must be separated by commas.

Execution part

Execution part contains program statements. There may or may not be a declaration part but there must be an execution part. It includes input and output statements.

Sub program section

Sub program section consists of all user-defined functions. It is called from the main function. It is placed before or after the main function. But generally placed after the main function.

C CHARACTER SET

Character set denotes alphabet, digit or special character. Characters combine to form variables. Characters in C are grouped into Letters, Digits, Special characters and White spaces. Compiler generally ignores white space when it is not a part of string constant. White Space may be used to separate words, but not used between characters of keywords or identifiers.

The character set in C Language can be grouped into the following categories.

1. Letters 2. Digits 3. Special Characters 4. White Spaces

Alphabets

Letters	**Letters**	**Digits**
Upper Case A to Z	Lower Case a to z	0 to 9

White Space Characters

It is used together with input and output statements. It has no meaning but it has the control to decide the way an input has to be displayed.

1. Blank Space	**\b**	2. Horizontal Tab	**\t**
3. Carriage Return	**\r**	4. New line	**\n**
5. Form Feed	**\f**	6. Vertical tab	**\v**

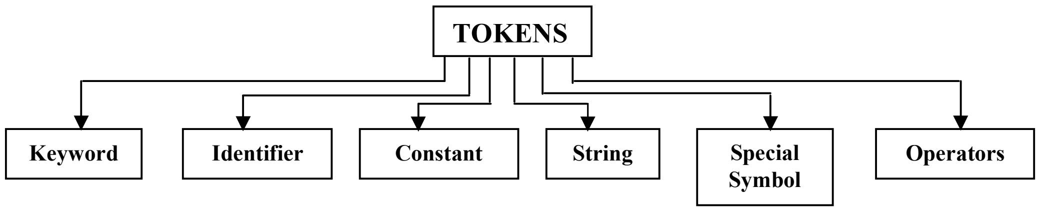
Special Characters

&	.Ampersand	#	.Number Sign
'	.Apostrophe	<	.Opening Angle (Less than sign)
*	.Asterisk	.	.Period (Dot)
@	At symbol	%	.Percentage Sign
\	.Backslash	+	.Plus Sign
^	.Caret	?	.Question Mark
>	.Closing Angle (Greater than sign)	"	.Quotation Marks
:	.Colon	}	.Right Flower Brace
,	.Comma	)	.Right Parenthesis
$	.Dollar Sign	]	.Right Bracket
. =	Equal to	;	.Semicolon
!	.Exclamation Mark	/	.Slash
(	.Left Parenthesis	~	.Tilde
[	.Left Bracket	_	.Underscore
{	.Left Flower Brace	|	.Vertical Bar

TOKENS

Among the group of text individual word, punctuation marks are called Tokens. It is a smallest individual unit in a C program. C has 6 types of Tokens. They are

- Keyword
- Identifier
- Special symbols
- Constant
- String
- Operators.



Keywords

Every word in C language is a keyword or an identifier. Keywords in C language cannot be used as a variable name. The compiler specifically uses them for its own purpose and they serve as building blocks of a c program. The following are the Keyword set of C language.

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

Identifiers

Identifier is a name given to program elements such as variables, functions, procedure, arrays and soon. First character must be an Alphabet or Underscore. Identifier consists of sequence of letters, digits or combination of both.

Rules for Identifier

- First character must be an Alphabet or Underscore(_).
- Special characters and embedded commas are not allowed.
- First 31 characters are given high priority or preference.
- Keyword cannot be used as Identifier.
- There should not be any white space.
- Both uppercase and lowercase letters are permitted.
- The underscore character is also permitted in identifiers.

CONSTANT

Constant is a fixed value that doesn’t or does not change during the execution of a program. It remains same throughout the program. Its value cannot be altered or modified in a program. A variable is declared by the Keyword const.

Eg: const int a,b,c;

Constant is classified into three main types. They are described as follows.

1. Numeric Constant
2. Character Constant and

3. Symbolic Constant.

NUMERIC CONSTANT

Numeric Constant is further classified into two main types. They are

(i) Integer Constants and (ii) Real Constants

Integer Constants

Integer constant consist of sequence of digit without decimal point. It is normally a whole number.

Real Constants

Real Constants are otherwise known as Floating Point Constants. Real Constant consists of decimal number, exponent or combination of both. Real constant consist of two forms. They are Fractional form and Exponential form.

Character Constant

Character Constant is also known as single character constant.

RESERVED WORD

Reserved words are inbuilt and have specific meaning. The reserved words used in C language are as follows.

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

VARIABLE

Variable is a name given to Memory location in which data is stored. Variable acts as value, which changes during the execution of program. Variable may take different value at different times during execution.

The general format of any declaration

datatype v1, v2, v3, ……….. vn

where v1, v2, v3 are variable names. Variables are separated by commas. A declaration statement must end with a semicolon.

Eg:
```
int sum;
double average, mean;
```

VARIABLE ASSIGNMENT

It is another format of variable declaration. The general form is

data-type variable name [=value];

Here data type refers to the type of value used in the C program. Variable name is a valid name. Square brackets [ ] denotes that the value may or may not be used or specified within the program.

Eg: int a; (or) int a=10;

Here if the value of a is assigned 10 the value remains the same throughout the program. If no value is specified then the value keeps on changing.

SIMPLE VARIABLE ASSIGNMENT

To assign a single value to a variable '=' equality operator is used. The syntax is

variable-name = value;

Eg: X=10; y=30;

COMPOUND VARIABLE ASSIGNMENT

variable-name = expression

These type of variable declaration can be used together with combination of characters, numbers or an expression. Here '=' equality operator is used.

Eg: Y=x+5; Z=((x+1)(y-2)*x)

Declaring a variable as constant

Variable is declared as constant by using the keyword or qualifier '**const**' before the variable name. This can be done at the time of initialization.

Eg: Const int class_size = 40;

Volatile Variable

A variable is volatile if the value gets changed at any time by some of the external sources.

Eg: volatile int num;

when we declare a variable as volatile the compiler will examine the value of the variable each time it is encountered to see if an external factor has changed the value.

DATA TYPES

Data Type is used to define the type of value to be used in a Program. Based on the type of value specified in the program specified amount of required Bytes will be allocated to the variables used in the program. Data types are broadly classified into three main types. They are as follows.

- Primary data type (Fundamental Data Types)
- Derived data type
- User defined data type.

Primary Data Type

Integers are represented as int, character are represented as char, floating point value are represented as float, double precision floating point are represented as double and finally void are primary data types. Primary data type offers extended data types. longint, long double are extended data types.

Integer Data Type

Integer data type can store only the whole numbers.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Integer	int	2	-32768 to 32767	%d
Short Integer	short int / short	2	-32768 to 32767	%d
Long Integer	long int / long	4	-2147483648 to 2147483647	%ld
Signed Integer	signed int	2	-32768 to 32767	%u
Unsigned Integer	unsigned int	2	0 to 65535	%d
Signed Short Integer	unsigned short int / short	2	-32768 to 32767	%d
Unsigned Short Integer	unsigned short int / short	2	0 to 65535	%u
Signed Long Integer	signed long int / long	4	-2147483648 to 2147483647	%ld
Unsigned Long Integer	unsigned long int / long	4	0 to 4294967295	%lu

Floating Point Data Type

Floating Point data types are also known as Real Numbers. It can store only the real numbers (decimal numbers) with 6 digits precision.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Float	float	4	3.4 e-38 to 3.4 e+38	%f
Double	double	8	1.7e-308 to 1.7e+308	%f
Long Double	long double	10	3.4 e-4932 to 3.4 e+4932	%lf

Character Data Type

Normally a single character is defined as char data type. It is specified by the keyword char. Char data type uses 8 bits for storage. Char may be signed char or unsigned char.

Name	C Representation	Size (bytes)	Range	Format Delimiter
Character	char	1	-128 to 127	%c
Signed Character	signed char	1	-128 to 127	%c
Unsigned Character	unsigned char	1	0 to 255	%c

Empty Data Type

VOID is the empty data type. This data type is used before main function in a C Language. The word void refers no return data type. It is used before the main function to specify the type of function. If a function is of type void it does not return any value to the calling function.

OPERATORS

Operator is a Symbol that tells or instructs the Compiler to perform certain Mathematical or Logical manipulations (Calculations). Operators are used in a program to work on data and variables. Operators in general form a part of Mathematical or Logical expression.

Operators are generally classified into different types. They are described one below another as follows.

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

ARITHMETIC OPERATORS

Arithmetic operators are used to perform Arithmetic operations. They form a part of program. Programs can be written with or without operators. But calculations are performed only using operators. Operators act along Operand to produce result.

Operator	**Meaning**	**Details**
+	Addition	Performs addition on integer numbers, floating point numbers. The variable name which is used is the **operand** and the symbol is **operator.**
-	Subtraction	Subtracts one number from another.
*	Multiplication	Used to perform multiplication
/	Division	It produces the Quotient value as output.
%	Modulo	It returns the remainder value as output.

Examples of arithmetic operators are

x + y x - y x * y x/y x%y

Here x, y are known as operands. The modulus operator is a special operator in C language that evaluates the remainder of the operands after division.

RELATIONAL OPERATOR

Relational Operators are used to compare two same quantities. There are six relational operators. They are mentioned one below another as follows.

Operator	**Meaning**	**Operator**	**Meaning**
<	is less than	>=	is greater than or equal to
<=	is less than or equal to	= =	is equal to
>	is greater than	!=	is not equal to

The general form is

(exp1 relational operator exp2)

where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25 TRUE -65 > 0 FALSE 10 < 7 + 5 TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

LOGICAL OPERATORS

Logical Operators are used when we need to check more than one condition. It is used to combine two or more relational expressions. Logical Operators are used in decision making. Logical expression yields value 0 or 1 i.e.,(Zero or One) . 0 indicates that the result of logical expression is TRUE and 1 indicates that the result of logical expression is FALSE.

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

Logical AND (&&)

The result of the Logical AND operator will be TRUE If both value is TRUE. If any one of the value is false then the result will be always False. The result is similar to basic Binary multiplication.

Eg: a > b && x = = 10

The expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (||)

If any of the expression is true the result is true else false otherwise. The result is similar to basic Binary addition.

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Eg: a < m || a < n

It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)

It acts upon single value. If the value is true result will be false and if the condition is false the result will be true. The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true.

Eg (!a)

EQUALITY OPERATOR

Equality operator (=) is used together with condition. The value of the expression is one or zero. If the expression is true the result is one, if false result is zero.

Operator	**Meaning**
= =	Equal to
!=	Not equal to

Example :

x = 1 and y = 2 then

x = =2 is false , x = = 1 true , y != 3 true

ASSIGNMENT OPERATORS

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression. The general form is

identifier = expression

Eg: x = a + b;

Here the value of a + b is evaluated and substituted to the variable x. '=' equal to is used in Assignment operators. It is of two types. They are

(i) Simple assignment operator

In Simple assignment operator only one '=' equal to operator is used. It is used to assign a value to variable or expression. The general form is

identifier = expression

Eg: y=35; x=ax+b-c;

(ii) Shorthand assignment operator

Shorthand assignment operator must be an arithmetic operator or bitwise operator. The general form is

identifier <operator> = expression

The operator must be an arithmetic operator or bitwise operator.

Operator	**Meaning**	**Example Simp. Assign**	**Example Shorthand**
+=	Assign sum	x = x + 1	x+ =1
-=	Assign difference	y = y - 1	y-=1
*=	Assign product	z=z*(x+y)	z*=(x+y)
/=	Assign quotient	y=y/(x+y)	y/=(x+y)
%=	Assign remainder	x=x%z	x%=z
~=	Assignone's complement		
<<=	Assign left shift	x = x << z	x << = z
>>=	Assign right shift		
&=	Assign bitwise AND	y = y&x	y&=x

|=	Assign bitwise OR		
^=	Assign bitwise X - OR	z = z^y	z^=y

CONDITIONAL OPERATORS

Conditional Operator Ternary operator is also known as “Ternary operator”. The general form of Conditional Operator is

(exp1)?(exp2):(exp3);

where exp1 is the condition which is to be checked and exp2 is the true value and exp3 is the false value. If the condition in exp1 is false then statement in exp3 will be automatically executed.

Eg:

```
#include<stdio.h>
void main()
{
        int x,y,z;
        clrscr();
        printf(“Enter the value of a and b :”);
        scanf(“%d %d”,&x,&y);
        z=((x>y)?x:y);
        printf(“The biggest value is %d”,z);
        getch();
}
```

Output:

```
Enter the value of a and b: 125 100
The biggest value is 125

Enter the value of a and b: 25 100
The biggest value is 100
```

SPECIAL OPERATORS

Special operators are known as separators or punctuators. Special operators are

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

Ampersand (&)

It is also known as addresss operator. It is specified before the identifier name. i.e., variable name. It indicates memory location of the identifier.

Asterisk (*)

Asterisk (*) is also known as indirection operator. It is specified before identifier name. it indicates creation of pointer variable. It is also a unary operator.

Braces ({ })

The opening brace (**{**) and closing brace (**}**) specify the start and end of compound statement in a function. Here semicolon (**;**) is not necessary. Semicolon (**;**) is used only in structure declaration.

Brackets

Brackets [] also referred as array subscript operator. It is used to indicate single and multi dimensional arrays.

Eg: int x[10]; float l[10][20];

Colon (:)

Colon (**:**) is used in labels. It is used in unconditional control statement i.e., in goto statement.

Eg: goto d;

Comma Operator (,)

It is used to link expressions together. It is used together with variables to separate one variable from another. It is used in for loop. It has the lowest precedence among operators

Eg:
```
for(n=1,m=10;n<=m; n++, m++)
int a,b,c;
sum= (x=5,y=3,x+y);
```

Ellipsis (…)

Ellipsis (…) are three continuous dots with no white spaces between them. It is used in function prototype. It indicate that the function can have any number of arguments.

Eg: void fun(char s,int n, float f, …);

Hash (#)

Hash **(#)** is also known as pound sign. it is used to indicate preprocessor directives.

Eg: #include“stdio.h”

Parenthesis (())

Parenthesis **(())** is also known as function call operator. It is used to indicate the open and end of function prototypes, function call, function parameters, …. Parenthesis are used to group expressions.

Semicolon (;)

Semicolon **(;)** is a statement delimiter. It is used to end a C statement.

Eg: g=d+h;

BITWISE OPERATORS

Bit wise operator is used to manipulate with bits within a word of memory. Bit wise operator operates only on integer and character but not on float and double.

Operator	Meaning	Operator	Meaning
~	One's Complement	&	Bitwise AND
<<	Left shift	|	Bit wise OR
>>	Right shift	^	Bit wise x-or

One's Complement Operator (~)

One's Complement makes the bits of operand inverted. Here one becomes zero and zero becomes one.

Eg: x = 7 i.e., x = 0000 0111

One's complement of t is 248 (i.e., ~x = 1111 1000 = 248)

Left shift Operator (<<)

Left shift operator (<<) shifts each bit of the operand to left. The general form is

Variable << number of bit positions

Eg: x =7 (i.e., 0000 0111 = 7)

y = x <<1 is 14. (i.e., 0000 1110 = 14)

Right shift Operator (>>)

Right shift operator shifts each bit of the operand to right. The general form is

Variable >> number of bit positions

Eg: x = 7 (0000 0111 = 7)

y = x >> 1 is 3 (i.e., 0000 011 = 3)

SIZE-OF OPERATOR

This operator is used to return the size of string or character. It cannot be used in together with Integers. The syntax is *sizeof(variable-name);*

Eg:

```
#include<stdio.h>
void main()
{
        int x;
        clrscr();
        printf("The value of x is %d",sizeof(x));
        getch();
}
```

Output:

```
The value of x is 2
```

PRECEDENCE OF OPERATORS

- Outermost Parenthesis is evaluated first.
- Then innermost parenthesis.
- If there is two or more parenthesis, then the order of execution is from left to right.
- Next Multiplication and Division are performed.
- Finally Addition and Subtraction.

INCREMENT and DECREMENT OPERATORS

Increment Operators

It is used to increase the value of the Operand by 1. There are two types of Increment Operators in C Language. They are pre Increment operator and post Increment operator.

Pre Increment Operator

It is used to increase the value of the variable by 1. Here the value of 1 is added to the variable first along with the given variable value.

Eg: ++g -> pre increment

Post Increment Operators

It is used to increase the value of the variable by 1. Here the value of 1 is added to the variable first along with the given variable value.

Eg: g++ -> post increment

Decrement Operators

It is used to decrease the value of the Operand by 1. There are two types of Decrement Operators in C Language. They are pre decrement operator and post decrement operator.

Pre decrement Operator

It is used to decrease the value of the variable by 1. Here the value of 1 is subtracted to the variable first along with the given variable value.

Eg: --g ------> pre decrement

Post Decrement Operator

It is used to decrease the value of the variable by 1. Here the value of 1 is subtracted to the variable first along with the given variable value.

Eg: g-- ----> post Decrement

UNARY OPERATORS

Unary operators act on single operand to produce new value. It precedes with operands. Unary minus denotes subtraction. Subtraction operators require two operands but unary minus require one operand. The operand used with this operator must be a single variable.

Operator	**Meaning**
-	Unary minus
++	Increment by 1
--	Decrement by 1
sizeof	Return the size of operand

Eg:

-786 -0.64 -5e-8 -(a+b) -6*(f+b) -45.878

When operator is used before variable then it is prefix notation. When operator is used after variable then it is postfix notation.

Eg:

Expression	**Result**
++x	4
x++	3
--x	2
x--	3

INPUT OUTPUT FUNCTIONS

Input Functions

Input Functions are used to accept data from user. It is used to transfer the value to the processor memory. The input/output functions are collectively known as Standard I/O Library. All input/output operations are carried out through function calls.

The commonly available Input functions are

- getchar()
- getc()
- gets()
- scanf()

getchar()

The getchar function can be used to read a character from the standard input device. i.e., keyboard. The general form of getchar() is

variable_name = getchar()

Here variable name is a valid name of type char.

Example program

```
# include < stdio.h >              // assigns stdio.h header file to your program
void main ( )                     // Indicates the starting point of the program.
{
        char C,                              // variable declaration
        printf (“Type one character:”) ;     // message to user
        C = getchar () ;     // get a character from key board and Stores it in variable C.
        printf (” The character you typed is = %c”, C) ; // output
}       // Statement which displays value of C on Standard screen.
```

getc()

This input function is used to read a character from the Input Unit i.e., Keyboard. The general form of getc() is

getc(stdin)

Here stdin refers to File Pointer from which Input is to be taken.

gets()

This input function is used to accept a string as Input from the user through the input device i.e., Keyboard. The function gets accepts the name of the string as a parameter, and fills the string with characters that are input from the keyboard till new line character is encountered.

The gets function relieves the string from standard input device. The standard form of the gets function is

gets(str)

Here str is a string variable.

Example program (Involving both gets and puts)

```
# include < stdio.h >
void main ( )
{
char s[80];            printf ("Type a string less than 80 characters:");
gets(s);               printf ("The string typed is:%s",s);   //   puts(s);
}
```

scanf()

This input function is used to provide input or accept input from the user in a program. It uses Format Specification String and list of variables as parameters. Format Specification used to define the type of data used as Input. The syntax for scanf() is

scanf("control string",argument list); *(or)*

scanf("cs1, cs2, ……csn",arg1, arg2,….argn);

Here Control String specifies the format in which the variables values provided as Input. Argument List specifies the address of Memory Locations where the values are being stored. Control String and Argument list are separated by Commas.

Format String	Meaning
%c	Read single character.
%d	Read a decimal integer.
%e	Read a floating point value in Exponential form.
%f	Read a floating point value.
%g	Read a floating point value.
%h	Reads a short integer.
%i	Read a decimal, hexadecimal or octal integer.
%o	Reads an octal integer.
%s	Reads an string.
%u	Reads an unsigned decimal integer.
%x	Reads an Hexadecimal integer. (Unsigned) using lower case a – f

%[..]	Reads a string of word(s).
%X	Reads a hexadecimal integer (Unsigned) using upper case A – F
%u	Reads a unsigned integer.
%U	Reads a unsigned long integer.
%p	Reads a pointer value
%hx	Reads hex short
%lo	Reads octal long
%ld	Reads long

Format String is followed by a prefix percentage (%) together with specific character.

Example:

For integer with d as value to be input %d. scanf (“%d %d”, &sum1, &sum2);

For character with c as value to be input %c. scanf (“%c %c”, &ch, nname):

Difference between scanf and printf function

Scanf	Printf
Used to accept data	Used to display data
Control string and & operator is used	Control string only used.
It end with semicolon	It end with semicolon
Number of input specified with format String and enclosed within double quotes.	Number of input specified with format string and separated by commas.
The Input variables are specified using Address operator (&) is separated by commas.	The output variables are specified by their name and separated by commas.

OUTPUT FUNCTIONS

Output Functions are used to transfer output data from processor memory to the output terminal. The output is normally seen in the Monitor. Output Functions are used to display result.

The commonly available Output functions are

- putchar()
- putc()
- puts()
- printf()
- clrscr()

putchar()

This function prints a single character on the standard output device called Monitor. This function has a single character as argument. This character is specified within single quotes. The general form of putchar() is

putchar(variable-name);

Here variable name is of type char and it contains of single character.

Eg:

```
#include < stdio.h > // Inserts stdio.h header file into the Pgm
void main ( )              // Beginning of main function.
{
        char in;                    // character declaration of variable in.
        printf(" please enter one character");            // message to user
        in = getchar ( ) ;                            // assign the keyboard input value to in.
        putchar(in);                         // out put 'in' value to standard screen.
}
```

clrscr()

This function is used together in a program. It is mentioned below variable declaration and above program statements. It helps to clear or remove the previously loaded values on the screen. Due to this whenever we run the program the previously run program results will not appear.

putc()

This function is used to write a single character from the input unit. It displays single character in the Monitor as output.

puts()

This function is used to display a string as output to the Output unit.

printf()

This function is used to print Numerical values i.e.,(integer, float, double), single characters, strings or combination of both. printf() function is used to redirect data from Computer's memory to standard output device. For float data type, number can be rounded to nearest decimal places.

The general form of printf() is

printf("control string", arg1, arg2, ..., argn); *(or)*
printf("control string", argument list);

Control string consists of characters that will be printed on the screen. Escape sequences and Format specification can be used together with the control string. Format specification is used to define the output format for each item, which is being displayed.

```
#include < stdio.h >
main ( )
{
printf("Hello!");
printf("Welcome to the world of Engineering!");
}
```

Output:

```
Hello! Welcome to world of Engineering.
```

UNIT - II

CONTROL STATEMENTS

The statements which are helpful in controlling the flow of execution are known as control statements. Control statements are mainly classified into two types. They are

- Unconditional Control statement
- Conditional Control statement.

UNCONDITIONAL CONTROL STATEMENT

The Execution of program can be altered by transferring the control from one place to another in a program without specifying the condition. This statement is known as Unconditional Control statement. GOTO is one of the Unconditional Control statement.

GOTO Statement

Conditional GOTO statement is used to transfer the control from one part of the program to other under certain condition. The general form is given below:

goto label

where **label** is an identifier.

Example program

```
#include<stdio.h>
void main()
{
        int age;
        clrscr();
        printf("Enter the Age :");
        scanf("%d",&age);
        if(age>=18)
        {
                goto m;
        }
        else
        {
                printf("\n You are not Eligible to vote:");
                exit();
        }

        m:      printf("You are Eligible to vote");
                getch();
}
```

I/P and O/P

```
Enter the Age: 18
You are Eligible to vote
```

I/P and O/P

```
Enter the Age : 10
You are not Eligible to vote
```

CONDITIONAL CONTROL STATEMENT

Those statements, which is used to check the Condition and based on the given condition carry out specific operations. Such type of statements is known as Conditional Control statement. Conditional Control statement is mainly of three main types. They are

(i) Conditional execution
(ii) Looping
(iii) Selection

CONDITIONAL EXECUTION

This statement executes the given condition based on the logical test. It performs two actions based on the result. The final result of the statement may be true or false. Some of the conditional execution statements are given below:.

- Simple IF statement
- IF…ELSE statement
- Nested IF..ELSE statement
- ELSE IF LADDER

SIMPLE IF STATEMENT

It statement is used in decision making execution. The condition part should not end with a semicolon.

The If structure has the following syntax

```
if(condition)
{
        Statement(s);
}
        statement-x;
```

If statement executes if the given condition is true otherwise it does not produces any output. If the condition is true statement will be executed otherwise the statement is skipped to statement-x.

Calculate the absolute value of an integer

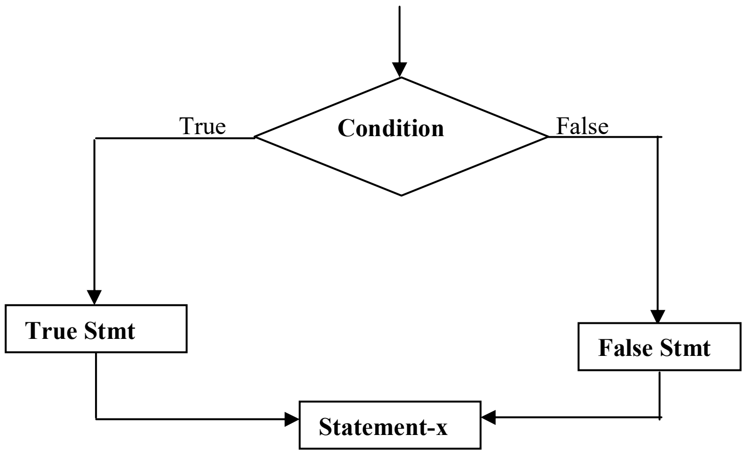
```
# include < stdio.h >           //Include the stdio.h file
void main ( )                   // start of the program
{
        int number;                     // Declare the variables
```

```
            clrscr();                         // clear the screen
            printf ("Type a number:");        // message to the user
            scanf ("%d", & number);           // read the number from standard input
            if (number < 0)                   // check whether the number is a negative
                    number = - number;        // If it is negative then convert it into positive.
            printf ("The absolute value is % d \n", number);  // print the value
            getch();
    }
```

IF..ELSE STATEMENT

If…else statement executes based on the given logical condition. If the result of the condition is true, then program statement 1 is executed, otherwise program statement 2 will be executed. The general form of if..else statement is as follows.

```
if(condition)
{
        true statement1;
}
else
{
        false statement2;
}
statement-x;
```



In these statement both if part and else part will not be executed at the same time. Either anyone of the both will be executed during run time.

// Program find whether a number is negative or positive */

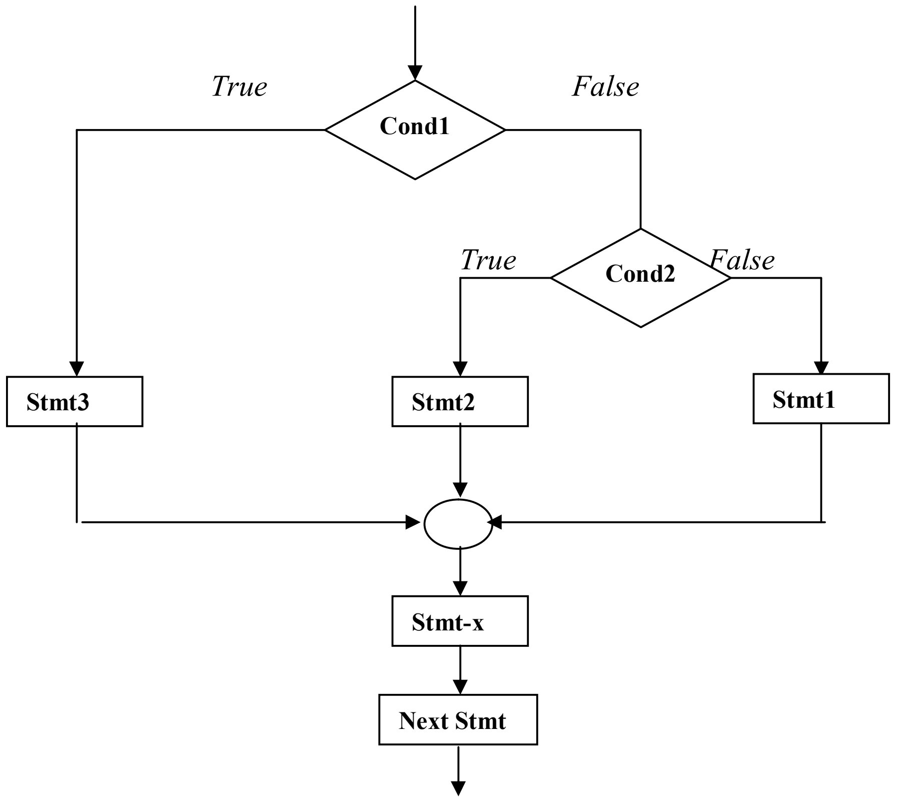
```
#include < stdio.h >              //include the stdio.h header file in your program
void main ( )                     // Start of the main
{
        int num;                          // declare variable num as integer
        printf ("Enter the number");   //message to the user
        scanf ("%d", &num);               // read the input number from keyboard
        if (num < 0)                      // check whether number is less than zero.
                printf ("The number is negative")      // If it is less than zero then it is negative.
        else                                            // else statement.
                printf ("The number is positive");     //If it is more than zero then it is +ve.
}
```

NESTED IF…ELSE STATEMENT

It is used to check with more than one Condition. If condition1 is true condition2 will be checked and statement1 will be executed followed by statement2 will be executed.

If condition1 is false statement3 will be executed. Otherwise default statement will be executed. The General form of Nested IF..Else statement is as follows.

```
if(condition1)
{    if(condition2)
     {
              statement1;
     }
     else
     {
              statement2;
     }
}
else
{
              statements;
}
statement-x;
}
```



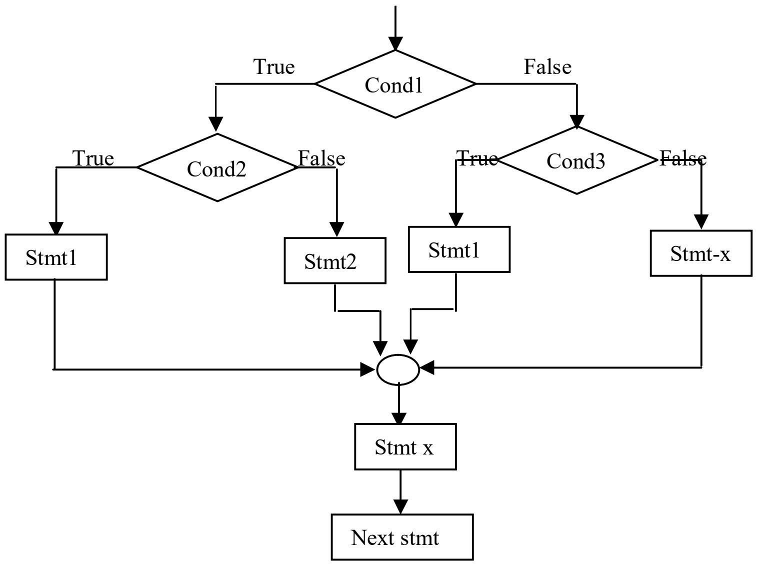
//Nested – If..Else

```
#include<stdio.h>
#include<conio.h>
void main()
{
        float a,b,c;
        clrscr();
        printf("Enter a Value");
        scanf("%f",&a);
        printf("Enter b Value");
        sanf("%f",&b);
        printf("Enter c Value");
        scanf("%f",&c);
        if(a>b)
        {
                if(a>c)
                        printf("%f\n",a);
                else
                        printf("%f\n",c);
        }
        else
{
        if(c>b)
                printf("%f\n",c);
        else
                printf("%f\n",b);
}
getch();
}
```

NESTED IF STATEMENT

If more than one if else statement is used then it is nested if-else statement. The if statement may itself contain another if statement is known as nested if statement. The general form of nested if..else statement is as follows.

```
if(cond1)
        if(cond2)
                statement-1;
        else
                statement-2;
else                            statement-3;
……………
else
statement x;
```



If more than one if else statement is used then it is nested if-else statement. The if statement may itself contain another if statement is known as nested if statement

//print the given numbers along with the largest number.

```
#include < stdio.h >     //includes the stdio.h file to your program
void main ( )            //start of main function
```

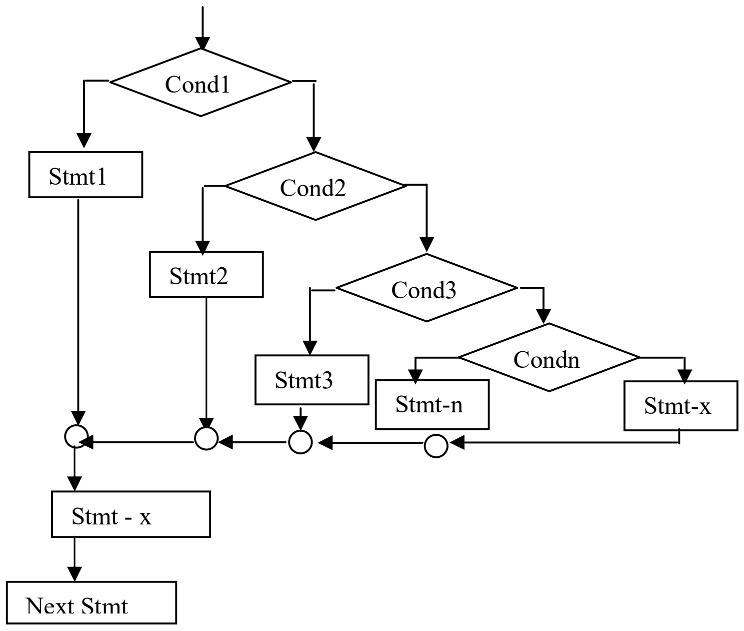
```
    {
            int a,b,c,big;              //declaration of variables
            printf ("Enter three numbers");                 //message to the user
            scanf ("%d %d %d", &a, &b, &c);    //Read variables a,b,c,
            if (a>b)                           // check whether a is greater than b if true then
                    if(a>c)                            // check whether a is greater than c
                            big = a ;                          // assign a to big
                    else big = c ;                     // assign c to big
            else if (b>c)    // if the condition (a>b) fails check whether b is  greater than c
                    big = b ;                          // assign b to big
            else big = c ;                     // assign C to big
            printf ("Largest of %d,%d&%d = %d", a,b,c,big);
            getch();
    }
```

ELSE IF LADDER

If multiple decisions are involved in if statement elseif ladder is used. Here condition1 is checked first. If it is true statement1 will be executed. Otherwise it checks for condition2. If condition2 is true statement2 will be executed. Otherwise it checks for condition-n and statement n will be executed. If all of the given condition becomes false then statement-x will be executed. The General form of elseif statement is as follows.

```
    if(condition1)
            statement1;
    elseif (condition2)
            statement2;
    elseif (conditon3)

            statement3;
            …………………
    elseif (condition)
            statement-n;
    else
            default statement:
            statement-x;
```



```
#include<stdio.h>
void main()
{
        int day;
        clrscr();
        printf("Enter a number between 1 to 7\n");
        scanf("%d",&day);
        if(day==1)
                printf("Monday\n");
        else if(day= =2)
                printf("Tuesday\n");
        else if(day= =3)
                printf("Wednesday\n");
        else if(day= =4)
                printf("Thursday\n");
        else if(day= =2)
                printf("Friday\n");
        else if(day= =2)
                printf("Saturday\n");
        else if(day= =2)
                printf("Sunday\n");
        else
                printf("Enter a Correct Number\n");
        getch();
}
```

SWITCH CASE STATEMENT

It is also known as selection statement. It helps to make decision from number of cases. The general form of Switch..Case statement is

```
switch(expression)
{
        case 1:
                statement;
                break;
        case 2:
                statement;
                break;
        …………
        default:
                statement;
}
```

//switch statement – add, sub, mul, div

```
#include<stdio.h>
void main()
{
        int a,b,choice;
        clrscr();
        printf("Enter two numbers : ");
        scanf("%d%d",&a,&b);
        printf("\n Enter 1 for addition  :");
        printf("\n Enter 2 for subtraction  :");
        printf("\n Enter3 for multiplication :");
        printf("\n Enter 4 for division :");
        printf ("\n Enter your choice :");
        scanf("%d",&choice :");
        switch (choice)
        {
           case1:
                printf("Sum is %d",a+b );               break;
           case2:
                printf("difference is %d",a-b);         break;
           case3:
                printf("Product is %d",a*b);            break;
           case4:
                printf("Quotient is %d",a/b);           break;
           default:
                printf("Invalid choice");
        }
        getch();
}
```

BREAK STATEMENT

Break statement is used to terminate the control from one place to another in a program. Break statement exits only single loop. Hence it is used together with the statements wherever necessary.

It is also used to exit from a loop. A break causes the innermost enclosing loop or switch to be exited immediately. It is also used together with while, dowhile, for and switch statement.

/* A program to find the average of the marks*/

```
#include < stdio.h >                  //include the stdio.h file to your program
#include<conio.h>
void main()                                  // Start of the program
{
        int i, num=0;                        //declare the variables and initialize
        float sum=0,average;   //declare the variables and initialize
        printf("Input the marks, -1 to end\n");        // Message to the user
        while(i)                                      // While loop starts
        {
                scanf("%d",&i);                              // read and store the input number
                if(i= =-1)                            // check whether input number is -1
                break;                                //if number –1 is input skip the loop
                sum+=i;                      //else add the value of I to sum
                num++;                                // increment num value by 1
        }
        getch();
}                                               // End of the program
```

CONTINUE STATEMENT

It is used to continue the process. Here keywords continue is used together within a statement. This statement is specified after the condition. The continue statement causes the loop to be continued with the next iteration after skipping any statement in between.

```
#include < stdio.h >               //Include stdio.h file
void main()                             //start of the program
{
        int i=1, num, sum=0;                  // declare and initialize the variables
        for (i = 0; i< 5; i++)                //for loop
        {
                printf("Enter the integer");  //Message to the user
                scanf("%i", &num);            //read and store the number
                if(num < 0)                    //check whether the number is less than zero
                {
                        printf("You have entered a negative number");   // message to the user
                        continue;             // starts with the beginning of the loop
                }                              // end of for loop
                sum+=num;                     // add and store sum to num
        }
        printf("The sum of positive numbers entered = %d",sum);    // print the sum.
}                                             // end of the program.
```

TYPE CONVERSION

Type Conversion is mainly of two types. They are

(i) Implicit type conversion (ii) Explicit type conversion

Implicit type conversion

Implicit type conversion used to combine constants and variables of different types in an expression. This automatic type conversion is known as implicit type conversion.

C Language automatically converts values to the proper type. If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type.

Rules for evaluating expressions

All short and char are automatically converted to int. Then

- If one operand is long double, the other will be converted to long double and result will be long double.

- If one operand is double, the other will be converted to double and result will be double.
- If one operand is float, the other will be converted to float and result will be float.
- If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.
- If one operand is long int and other is unsigned int then
  a. If unsigned int can be converted to long int, then unsigned int operand will be converted as such and the result will be long int.
  b. Else both operands will be converted to unsigned long int and the result will be unsigned long int.
- If one of the operand is long int, the other will be converted to long int and the result will be long int.
- If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

Explicit Conversion

Explicit conversion used to force a data type to convert into another data type.

The general form is

(type_name) expression;

Example:

Ratio = femstud / malstud

Here **femstud** and **mastud** are declared as integers, the decimal part will be rounded off. This problem can be solved by converting one of the variables to the floating point.

```
#include<stdio.h>
void main()
{
	int x,y;
	float z;
	clrscr();
	printf(“\n Enter the value of x:”);
	scanf(“%d”,&x);
	printf(“\n Enter the value of y :”);
	scanf(“%d”,&y);
	z=(float)((x+y)/(x-y));
	printf(“\n The result of the expression z is %f”,z);
	getch();
}
```

LOOPING STATEMENTS

The process of executing a statement (or) group of statement repeatedly until a particular condition is satisfied is called Looping.

Loop is nothing but the part of a program code that execute repeatedly is called Loop. There are three types of Looping statements. They are

(i) while loop
(ii) do-while loop
(iii) for loop.

WHILE LOOP

The general form of while statement is

```
while(test condition)
{
        statement(s);
}
```

while statement executes only if the given condition is true. While loop is also called entry-controlled loop. Entry-controlled loop also known as pre-test.

Example program for generating 'N' Natural numbers using while loop:

```
# include<stdio.h >              //include the stdio.h file
#include<conio.h>
void main()                      // Start of your program
{
        int n, i=0;                      //Declare and initialize the variables
        clrscr();
        printf("Enter the upper limit number");             //Message to the user
        scanf("%d", &n);                                 //read and store the number
        while(i < = n)                                   // While statement with condition
        {                                        // Body of the loop
                printf("\t%d\n",i);                       // print the value of i
                i++;            increment I to the next natural number.
        }
        getch();
}
```

DO..WHILE LOOP

In do..while loop , statement is executed first then the condition is checked. If the given condition is not true at-least one statement will be definitely executed. If the condition is true then the while loop will be executed.

The syntax of the do while loop is:

```
do
{
        statement;
}
while(expression);
```

do-While loop is also called exit-controlled loop. Entry-controlled loop also known as post-test.

/* Program to illustrate the do while loop*/

```
#include<stdio.h >                     //include stdio.h file to your program
#include<conio.h>
void main()                                  // start of your program
{
char inchar;                                 // declaration of the character
        clrscr();                                    //  clear the screen
        do                                           // start of the do loop
        {
        printf("Input Y or N");                      //message for the user
        scanf("%c", &inchar);                // read and store the character
        }
        while(inchar!='y' && inchar != 'n');         //while loop ends
        if(inchar=='y')                   // checks whther entered character is y
        printf("you pressed u\n");                    // message for the user
        else
        printf("You pressed n\n");
        getch();
        }                                            //end of for loop
```

FOR LOOP

The for loop is an **Entry Controlled Loop**. It is a post test. The syntax of FOR loop is shown below:

```
for(initialization; test-condition; Increment/decrement)
{
        program statement(s);
}
```

Here initialization is done in the beginning of the for loop. Test condition is the place where the given condition is being checked. Condition is checked every time using the initialization value. Based on the condition initialized value will be incremented / decremented.

For loop example program:

```
/* The following is an example that finds the sum of the first fifteen positive natural
numbers*/
#include<stdio.h>                    //Include stdio.h file
#include<conio.h>
void main()                          //start main program
{
        int i;
        clrscr();                    //declare variable
        int sum=0,sumsqr=0;          //declare and initialize variable.
        for(i=0;i< = 30; i+=2)       //for loop
        {
                sum+=i;              //add the value of I and store it to sum
                sumsqr+=i*i;         //find the square value and add it to sumsqr
        }                            //end of for loop
        printf("Sum of first 15 positive even numbers=%d\n",sum);       //Print sum
        printf("Sum of their squares=%d\n",sumsqr);                     //print sumsqr
        getch();
}
```

NESTED FOR LOOP

Loops containing more then one loops within itself is known as nested loop. Here outermost loop will start to execute first but the process of outermost loop ends only when the condition of innermost loop ends. Each loop must have a unique index variable. Each loop should be embedded with each other. Loops can have any number of exit point but should not permit entry into it. Loops should not overlap each other.

//Pascaline Triangle

```
        #include<stdio.h>            #include<conio.h>
        void main( )
        {
                int i,j;  clrscr();
                for(i=1;i<=10;i++)
                {
                        for(j=1;j<=i;j++)
                        {
                                printf("*");
                        }
                        printf("\n");
                }
                getch();
        }
```

UNIT - III

ARRAY

Array is a group of elements of the same data type that share a common name. Each array element is specified by array name with one or more subscripts. Subscripts are enclosed within square brackets [ ]. Individual values in an array are called elements.

(or)

Array is a sequenced collection of related data items that share a common name and same data type. Each array element is specified by array name with index or subscripts in brackets after array name.

(or)

Array is a collection of identical data objects which are stored in consecutive memory locations in a common variable name. Individual values in an array are called elements.

ARRAY DECLARATION

In general, array is declared as

data-type arrayname[size]; or
data-type arrayname[subscript];

Data-type refers to a valid data type. Size refers to the maximum number of elements an array can hold at the maximum. Array must be declared before being used in the C program.

Example

```
float height[50];
```

// Program to print sum of ten numbers

```
#include<stdio.h>
void main()
{
        int i;
        float num[10] ,total=0.0;     /*read array values*/
        printf("enter 10 real numbers\n");
        for(i=0;i<10;i++)
        {
                scanf("%f",&num[i]);
                total+=num[i];          /*calculate total*/
        }
        for(i=0;i<10;i++)
        {
                printf("num[%d]=%5.2f\n",i,x[i]);
                printf("\n Total=%2f\n",total);
        }
        getch();
}
```

DIMENSIONING ARRAY

Declaring the name, type of the array and setting the number of elements in the array is known as dimensioning the array.

Array declaration helps to define the type of the array, name of the array, number of subscripts in an array and the total number of memory locations to be allocated.

```
#include< stdio.h >
#include<conio.h>
void main( )
{
        int a[50],n,count_neg=0,count_pos=0,i;
        clrscr();
        printf("Enter the size of the array\n");
        scanf("%d",&n);
        printf("Enter the elements of the array\n");
        for(i=0;i< n;i++)
                scanf("%d",&a[i]);
        for(i=0;i< n;i++)
        {
                if(a[i]< 0)
                        count_neg++;
                else
                        count_pos++;
        }
        printf("There are %d negative numbers in the array\n",count_neg);
        printf("There are %d positive numbers in the array\n",count_pos);
        getch();
}
```

One Dimensional Array Declaration.

The One Dimensional Array is generally declared as follows.

data-type arrayname[size];

Data-type refers to a valid data type. Size refers to the maximum number of elements an array can hold at the maximum. Array must be declared before being used in the C program.

//Add ten numbers and find sum and average

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,num[10],sum=0;
        float avg=0.0;
        clrscr();
        printf("enter ten numbers:\n");
        for(i=0;i<10;i++)
        {
                scanf("%",&num[i]);
                sum+=num[i];
        }
        avg=(float)sum/10.0;
        printf("\n sum of 10 number is        : %7.2d",sum);
        printf("\n average of 10 number is : %5.3f",avg);
        getch();
}
```

Array Initialization (Initializing an Array)

Array can be initialized in two ways. They are

(i) at compile time and

(ii) at run time.

At compile time

Array can be initialized at compile time. The general form to initialize an array at compile time is as follows. The general form is

data-type array-name[size]={value1, value2,……… , value n};

Data-type refers to the type of value to be used in a C program. Array name refers to a valid array name. Size refers to the maximum number of elements an array can hold at the maximum. Character string terminated by a null character '\0'.

The values in the array may be one, two, three and so on (i.e.,) 1,2,3,…………n. Here n specifies the number of values being used. Values are initialized from left to right. First value is assigned to arrays first position and second value is assigned to arrays second position and so on. Each values in an array are separated by commas(,) and terminated by close brace } followed by semicolon.

Example

```
int number[3]={0,0,0};
int counter[]={1,1,1,1};
char month1[ ]={‘j’,’a’,’n’,’u’,’a’,’r’,’y’};
```

//Program to read marks using array initialization

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int studmark[5]={99,97,87,89,92};
        int i;
        clrscr();
        printf(“mark of the student is:\n”);
        for(i=0;i<=4;i++)
        {
                printf(“\n Student Mark[%d]=%d\n”,i,studmark[i]);
        }
        getch();
}
```

//Program to read subject name using string initialization

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char SubName[5][20]={”Electron”,”Robotic”,”Mechanic”,”DOS”,”CAD”};
        int i; clrscr();
        printf(“Name of the Subject is:\n”);
        for(i=0;i<5;i++)
        {
                printf(“\t SubName[%d]=%s\n”,i,SubName[i]);
        }
        getch();
}
```

/* Program to count the no of positive and negative numbers*/

```
#include< stdio.h >
#include<conio.h>
void main( )
{
        int a[50],n,count_neg=0,count_pos=0,i;
        clrscr();
        printf(“Enter the size of the array\n”);
        scanf(“%d”,&n);
        printf(“Enter the elements of the array\n”);
        for (i=0;i< n;i++)
```

```
                scanf("%d",&a[i]);
                for(i=0;i< n;i++)
                {
                        if(a[i] < 0)
                                count_neg++;
                        else
                                count_pos++;
                }
        printf("There are %d negative numbers in the array\n",count_neg);
        printf("There are %d positive numbers in the array\n",count_pos);
        getch();
}
```

/*String.c string variable*/

```
#include < stdio.h >
#include<conio.h>
oid main()
{
        char month[15];
        clrscr();
        printf ("Enter the string");
        gets(month);
        printf ("The string entered is %s", month);
        getch();
        }
```

Run time array initialization

Array values can be initialized at run time. To initialize values at run time the n values specified as input is being compared with the condition and each time executed.

//Input n numbers and display n numbers

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[100], i,n;
        clrscr();
        printf("number of elements in array\n");
        scanf("%d",&n);
        printf("enter the elements\n");
        for(i=0;i<n-1;++i)
        {
                scanf("%d",&a[i])
        }
        printf("elements in array\n");
        for(i=0;i<=n-1;++i)
        {
                printf("%d\t",a[i]);
        }
        getch();
}
```

// copy contents to another string

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[10],s2[10];     int i=0;
clrscr();
printf("enter first string:"); gets(s1);
/*copy content of s1 to s2*/
while(s1[i]!='\0')
{
s2[i]=s1[i];            i++;
}  /*terminate second string*/
s2[i]='\0';
/*print the string*/
printf("the output is :\n");
puts(s2); puts(s1); getch();
}
```

Unsized Array.

If the size of the Array is unknown then it is said to be unsized Array. It is denoted by data-type followed by the array name and square braces. Here value is specified within square braces. The syntax is

data-type arrayname[ ] = { value 1, value 2,………………value n };

Here data-type refers to a valid C data type. Array name refers to name of the array. It refers to the maximum number of elements an array can hold at the maximum. Each value in an array are separated by commas and terminated by semicolon.

// Input 10 characters and print 10 characters

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char character[10];
        int cnt;
        clrscr();
        /*read character one by one*/
        printf(“enter 10 character\n”);
        for(cnt=0;cnt<10;cnt++)
                character[cnt]=getchar();
        /*display character one by one*/
        printf(“\nentered characters are :\n”);
        for(cnt=0;cnt<10;cnt++)
                putchar(character[cnt]);
}
```

//Count occurrence of desired character from a given string

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char s[80],c;
        int i=0,n=0;
        clrscr();
        printf(“\n enter the string:”);
        gets(s);
        printf(“\n enter the character to search:”);
        c=getchar();
        for(i=0;s[i]!=’\0’;i++)
                if(s[i]==c)
                n++;
        printf(“\n the character %c occurs %d times in %s”,c,n,s);
        getch();
}
```

Initializing Character

```
/* Character is initialized within single quotes.*/
        #include<stdio.h>
        #include<conio.h>
        void main()
        {
                char vow[5]={‘a’,’e’,’i'’,’o’,’u’};
                int i; clrscr();
                printf(“Vowels are:\n”);
                for(i=0;i<5;i++)
                {
                printf(“\t Vow[%d]=%c\n”,i,vow[i]);
                }
                getch();
        }
```

/*String.c string variable*/

```
#include < stdio.h >
void main()
{
        char month[15];
        clrscr();
        printf ("Enter the string");
        gets (month);
        printf ("The string entered is %s", month);
        getch();
}
```

SORTING AN ARRAY

Sorting is the process of Arranging the set of data items in an order. Sorting may be carried out in Ascending or Descending order. Sorting helps to increase the efficiency of a program.

//Sort number in ascending order

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j,k,num[5],temp;
        clrscr();
        printf("enter five numbers:\n");
        for(i=0;i<5;i++)
        {
                scanf("%d",&num[i]);
        }
        printf("\n THE ORIGINAL LIST IS:\n");
        for(i=0;i<5;i++)
        {
                printf("%d\t",num[i]);
        }
        for(i=0;i<4;i++)
        {
                for(j=i+1;j<5;j++)
                {
                        if num[i]>num[j]
                        {
                                temp=num[i];
                                num[i]=num[j];
                                num[j]=temp;
                        }
                }
        }
        printf("\n the sorted numbers are:\n");
        for(i=0;i<5;i++)
        {
                printf("%d\t",num[i]);
        }getch();
}
```

PASSING ARRAYS TO A FUNCTION

Array values are passed to a function as parameter. The name of the array is used as Argument to a function. When an array is passed to the function, address of first array element is passed to the function. Here array name must not be enclosed within brackets and it should not include subscripts.

```
#include<stdio.h>
#include<conio.h>
Int num[5];
void main()
{
        int i;
        clrscr();
        printf("enter five numbers:\n");
        for(i=0;i<5;i++)
        {
                scanf("%d",num[i]);
        }
        printf("\n the original list is:\n");
        for(i=0;i<5;i++)
                printf("%d",num[i]);
                sort(num);
        printf("\n the sorted list is:\n");
        for(i=0;i<5;i++)
                printf("\t%d",num[i]);
        getch();
}

sort(int num[])
{
        int i,j,temp;
        for(i=0;i<4;i++)
        {
                for(j=i+1;j<5;j++)
                if(n[i]>n[j])
                {
                        temp=n[i];
                        n[i]=n[j];
                        n[j]=temp;
                }
        }
return;
}
```

TWO DIMENSION ARRAY

Two dimensional array consist of (or) made up of two dimensions i.e., rows and columns. It contains two subscripts. First subscript is represented as ‘r’ which stands for the total number of rows and the Second subscript ‘c’ stands for number of columns.

A Two Dimension Array takes the general form as follows.

data-type array-name[r] [c];

Here r stands for number of rows and c stands for number of columns. Data-type refers to a valid C data type. Array-name refers to a valid array name.

If there are 3 rows and 3 columns then total number of values will be 3×3 values. In general for an array with n rows and c columns total number of values will be n×c values.

Rules for Declaring Two-Dimensional Array

- For matrix addition and subtraction first matrix row, column and second matrix row, column should be the same.
- For matrix multiplication, first matrix column and second matrix row must be equal.

/* example program to add two matrices & store the results in the 3rd matrix */

/***** MATRIX ADDITION ****/**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,j,m,n;
        int l[5][5],m[5][5],n[5][5];
        clrscr();
        printf("\n\t\t\t MATRIX ADDITION \n\n");
        printf("\n Enter the Order of Matrix: \n");
        scanf("%d%d",&m,&n);
        printf("\nEnter the Elements of 1st Matris:\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&m[i][j]);
                }
                printf("\n");
        }
        printf("\n Enter the Elements of 2nd Matrix: \n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&n[i][j]);
                }
                printf("\n");
        }
        printf("\n The Resultant Matrix is: \n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        l[i][j]=m[i][j]+n[i][j];
                        printf("%d\t",&c[i][j]);
                }
                printf("\n\n");
        }
        getch();
}
```

Initializing an Two-Dimensional Array.

A Two-Dimensional Array can be also initialized. For that the array values are specified within a Compound statement.

data-type array-name[r][c] = { value1, value2, ... , value n};

Data-type refers to a valid C data type. Array-name refers to a valid array name. Here r stands for number of rows and c stands for number of columns. Each value in an array are separated by commas and terminated by semicolon.

```
#include<stdio.h>
void main()
{
        int i,j;
        float num[4][2]={{12.3,34,5},{23.4,45.6},{34.5,56.7},{45.6,67.8}};
        printf("Element  -  value  -  Address");
        for(i=0;i<4;i++)
        {
                for(j=0;j<2;j++)
                {
                printf("\n num[%d] [%d] =  %0.2f \n  ",i,j,num[i][j]);
                }
        }
        getch();
}
```

MULTI-DIMENSIONAL ARRAY

Multi-dimensional Array consists of (or) requires more than two square brackets and it may contain any number of values specified within square brackets. It may be three, four, five, six and so on. A Multi dimensional Array in general takes the following form.

data-type array-name[s1][s2]……………[sn];

Data-type refers to a valid C data type. Array-name refers to a valid array name. s1,s2,s3, ……… are sub scripts.

```
#include<stdio.h>
void main()
{
        int mark[2][4][3],i,j,k;
        int totmark,semsum,papsum;
        clrscr();
        for(i=0;i<2;i++)
        {
                printf("For the %d semester \n",i+1);
                for(j=0;j<4;j++)
                {
                        printf("\n Enter 3 marks for paper %d\n",,j+1);
                        for(k=0;k<3;k++)
                                scanf("%d",&mark[i][j][k]);
                }
        }
        totmark=0;
        for(i=0;i<2;i++)
        {
                semsum=0;
                printf("\n Semester %d\n",i+1);
                printf("Paper  marks \ n");
                for(j=0;j<4;j++)
                {
                        papsum=0;
                        for(k=0;k<3;k++)
                                papsum+=mark[i][j][k];
                                printf("%d      %d\n",j+1,papsum);
                                semsum+=papsum;
                }
                printf("Total marks in %d semester = %d \n",i+1,semsum);
                totmark+=semsum;
        }
        printf("\n Grand total = %d",totmark);
        getch();
}
```

INITIALIZING MULTI-DIMENSIONAL ARRAY

Multi-dimensional Array consists of (or) requires more than two square brackets and it may contain any number of values specified within square brackets. It may be three, four, five, six and so on. A Multi dimensional Array can be also initialized. The general form is

data-type array-name[s1][s2]…[sn] = { value1, value2, … valuen};

Example

int num[2][3][2] = { 0,1,2,3,4,5,6,7,8,9,10,11};

Now here

num[0][0][0]=0;num[0][0][1]=1;num[0][1][0]=2;num[0][1][1]=3;num[0][2][0]=4;
num[0][2][1]=5;num[1][0][0]=6;num[1][0][1]=7;num[1][1][0]=8;num[1][1][1]=9;
num[1][2][0]=10;num[1][2][1]=11;

Example:

int table[2][3]={0,0,0,1,1,1}; (or) int table[2][3]={{0,0,0},{1,1,1}}
int survey[3][5][12]; float table[5][4][5][3];

FUNCTIONS

DEFINITIONS OF FUNCTIONS:

- ✓ Self-contained program segment that is placed separately from the main program to perform some specific well defined task is called **function**.
- ✓ **Function** is a small segment of program that carries out some specific and well-defined task.
- ✓ A program segment that is placed separately from the main program is called **function**.
- ✓ Functions are the building blocks of a C program.

Advantages:

- ✓ It reduces the length of source program.
- ✓ Breaks the complexity of a program.
- ✓ Break small program into small program.
- ✓ It is easy to maintain, modify and understand.
- ✓ A program can be divided into smaller subprograms.
- ✓ It facilitates top down modular programming.

- ✓ The length of the source program can be reduced using functions
- ✓ Critical in microcomputers, where memory space is limited.
- ✓ Avoid rewriting the same sequence of code at two or more locations in a program.

Objective of this chapter:

- ✓ How a function is designed?
- ✓ How a function is integrated into a program?
- ✓ How two or more functions are put together?
- ✓ How they communicate with one another?

FUNCTION CLASSIFICATION

Function in C Language is mainly classified into two types. They are as follows.

- ✓ Library functions (Built – in functions)
- ✓ User defined functions.

User defined functions are designed and developed by the user while writing and compiling a C program. (Example: Main function)

Library functions

Library functions are also known as Built-in functions. The Library functions available in C language are

(i) Input/Output functions.
(ii) Boolean functions.
(iii) Conversion functions.
(iv) Memory functions.
(v) String functions.
(vi) Miscellaneous functions.
(vii) Math or Arithmetic functions and
(viii) Time functions.

A MULTI-FUNCTION PROGRAM

- ✓ A function is a self contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. Every C program can be designed using a collection of these black boxes know as functions.

- ✓ **Consider a set of statements shown below:**

```
void display(void)
{
int i;
for(i=1;i<20;i++)
printf("-");
printf("\n");
}
```

This above set of statements defines a function called display, which could print a line of 20-character length. This function can be used in a program as follows:

- ✓ **Example 1:**

```
main()
{
display();
printf("Computer programming\n");
display();
}
void display(void)
{
int i;
for(i=1;i<20;i++)
printf("-");
printf("\n");
}
```

Output:

```
----------------------------
Computer Programming
----------------------------
```

- ✓ The main function calls the user defined function "display" two times and library function "printf" once. "Display" function itself calls the library function 19 times repeatedly.

- ✓ Any function can call any other function, In fact it can call itself. A “called function” can also call another function. A function can also be called more than once.

ELEMENTS OF USER-DEFINED FUNCTIONS

- ✓ User defined function has three elements. They are
  1. Function definition (function implementation)
  2. Function call
  3. Function declaration
- ✓ **Function definition** is an independent program module that is specially written to implement the requirements of the function. In order to use this we need to invoke it at a required place in the program. This is known as **function call**. The program that calls the function is referred to as the calling program or calling function. The calling program should declare any function that is to be used in the program. This is known as **function declaration**.

FUNCTION DEFINITION

- ✓ A function definition, also known as function implementation shall include the following elements.

1. Function name	Function Header	
2. Function type		….. may be int, float, double
3. List of parameters		
4. Local variable declarations	Function body	
5. Function statements		
6. Return statement		

- ✓ The general format of a function definition to implement these two parts is as follows.

```
function-type function-name(parameter list)
{
        local variable declaration;
        executable statement1;
        executable statement2;
        ……………………..
        return statement
}
```

- ✓ The first line indicates **function-type function-name (parameter list)** is known as the **function header** and the statements inside the braces constitute the **function body.**

FUNCTION HEADER

- ✓ It consists of three parts.
  1. function type(return type)
  2. function name
  3. formal parameter list

Function Type

- ✓ It specifies the type of the value (like int, float or double) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer. If the function is not returning anything, then we need to specify the return type as void.

Function name

- ✓ It is any valid C identifier that must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function.

Formal Parameter list

- ✓ The parameter list declares the variables that will receive the data sent by the calling program. They serve as the input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as formal parameters. These parameters can also be used to send values to the calling programs.

- ✓ The parameter list contains the declaration variables separated by commas and surrounded by parenthesis.

Examples:

int circle();{------}	//	No argument
int area(int p,int r);{------}	//	Two argument
int rectangle(int h, int b, int w);{---}	//	Three argument.
float quadratic(int a, int b, int c);{-----}	//	Three argument

FUNCTION BODY

- ✓ Function body contains declaration and statements necessary for that function. It contains compound statements. Function is called by its name together with list of arguments. When the function is invoked Formal Parameters are initialized to Actual Parameters. Function body contains declaration and statements necessary for that function. Function body is enclosed in braces.
- ✓ Function body consists of three main parts. They are
  1. Local declaration
  2. Function statements
  3. Return statements

Local declaration

- ✓ It is used to specify the variables necessary for a function. Variables declared inside the main function is called as Local variables and Variables declared outside the main function is called Global variables.

Function statement

- ✓ Function statement is used to perform task for the function.

Return statement

- ✓ 'Return' statement is used to return the processed value to the main function.
- ✓ If a function does not return any value to the calling function, we can omit the return statement, then its return type is 'void'.
- ✓ If a function does not receive any value from the sub function then there is no need of 'return' statement.
- ✓ Parameters are also known as Arguments.

RETURN VALUE AND THEIR TYPES

- ✓ A function may or may not send back any value to the calling function. If it does, it is done through the **return** statement. while it is possible to pass to the called function any number of values, the called function can only return one value per call.
- ✓ It is used to return a value to the calling program (or) function call. It is used to terminate a function and return a value to the calling program. The return statement may or may not contain an expression. The return statement can take one of the following forms

    return;

    or

    return(expression);

Example:

```
int mul(int x, int y)
{
int p;
p=x*y;
return(p);
}
```

returns the value of p which is the product of the values of x and y. The last two statements can be combined into one statement as follows

```
return(x*y);
```

- ✓ A function may have more than one return statements.

FUNCTION CALLS:

- ✓ A function can be called by simply using the function name followed by a list of actual parameter.

Example

```
main()
{
int y;
y=mul(8, 6);
printf("%d\n", y);
}
```

When a function is called, the control is transferred to the function mul(). This function is then executed and a value is returned when return statement is used. This value is assigned to y. This is illustrated below

```
main()
{
int y;
y=mul(8,6};
}
          int mul(int x, int y)
          {
          int p;
          p=x*y;
          return(p);
          }
```

The function calls sends two integer valued 8 and 6 to the function.

FUNCTION DECLARATION

✓ Like variables, all functions in a C program must be declared, before they are invoked. It was also known as Function prototype. It consists of four parts.

1. Function type (return type)
2. Function name
3. Parameter list
4. Terminating semicolon

General form:

datatype function name (argument list);

This is very similar to the function header line except he terminating semicolon.

Examples:

```
double sqroot(int num);
void prod(int a, int b);
int mul(int m, int n);
```

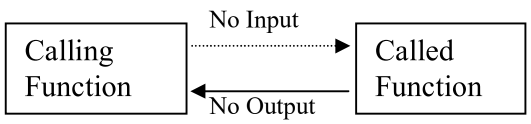
TYPES OF USER DEFINED FUNCTIONS

In general, a function based on whether the Argument is present or not, whether the value is returned or not is classified into five types. They are

(i) Functions with no Arguments and no return values.
(ii) Functions with Arguments and no return values.
(iii) Functions with no Arguments and return values.
(iv) Functions with Arguments and return values.
(v) Functions that return multiple values.

FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUES.

✓ When a function has no Argument it does not receive any data from the calling function. When a function does not return value, calling function does not receive data from the called function. There is no data transfer between calling function and called function. A function that does not return any value cannot be used in an expression.



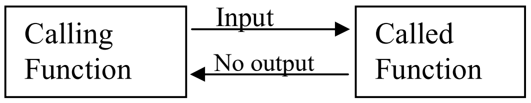
Example: Program to illustrate a function with no argument and no return values

```
#include<stdio.h>                    // Header file
#include<conio.h>
void main()                          // main function
{                                    // beginning of c program
        long i, j,x=1;               // variable declaration
        void square();               // Function declaration or function prototype
        clrscr();                            // clear the screen
        printf("\n Enter a number :"); // output the text
        scanf("%ld",&i);             // read number i
        for(j=1;j<=i;j++)
        x=j*x;
        printf("\nThe Factorial of %ld is %ld",i,x);
        square();                    // Function call
getch();
}
void square()
```

```
{
        int i,n,s,d,sum=0;
        printf("Enter the range:");      scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
        s=d*d;           sum+=s;
        printf("Square of %d is %d",i,s);
        }
printf("\n Sum of squares is %d",sum);
}
```

FUNCTIONS WITH ARGUMENTS AND NO RETURN VALUES.

- ✓ When a function has Argument it receives data from the calling function and the value is used inside the sub-function. Whenever a function call is made, a copy of the values in Actual argument are sent to the Formal argument by using the called function. A calling function sends value to arguments by means of function call.



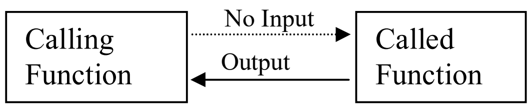
- ✓ The Actual arguments and Formal arguments must be the same in number, order and type. The values used in actual arguments must be assigned values before the function call is made.

Example: Program to find the largest of two numbers using function

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
void largest (a,b);
clrscr();
printf("Enter the two numbers");
scanf("%d%d",&a,&b);
largest(a,b) ;
getch();
}
```
/*Function to find the largest of two numbers*/
```
void largest(int a, int b)
{
if(a>b)
printf("Largest element=%d",a);
else
printf("Smallest element=%d",b);
}
```

FUNCTIONS WITH NO ARGUMENTS AND RETURN VALUES.

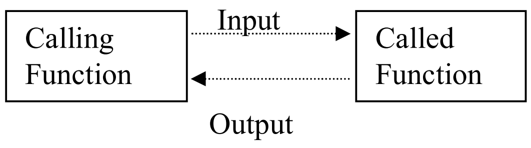
- ✓ When a function has no arguments it does not receive any data from calling function. But in called function some process takes place and value is returned to calling portion of main program. To return value to the main function **return** statement is used. Hence there is data transfer between calling function and called function.



Example: No Argument and return value
```
#include<stdio.h>
#include<conio.h>
void main()
{
int h;
void add();
clrscr();
h=add();
printf("The sum of  Numbers is:%d",h);
getch();
}
int add()
{
int j,k,l;
printf("Enter the value of j:");
scanf("%d",&j);
printf("Enter the value of k:");
scanf("%d",&k);
l=j+k;
return(l);
}
```

FUNCTIONS WITH ARGUMENTS AND RETURN VALUES.

- ✓ When a function has argument it receives data from calling function and does some process and return value to the called function. Here main function has control over the function. Whenever function is called, Actual argument values are copied and assigned to the Formal arguments. Now from the values in the formal arguments sub-function is processed and value is returned to the main function.



```
#include<stdio.h>
#include<conio.h>
void main()
{
float x,y;
clrscr();
float add(int x, int y);           // function declaration
double sub(int x,int y);           //function declaration or function prototype
x=12.345;
y=9.82;
printf("x=%f\n" add(x,y));         // function call – x,y are actual arguments
printf("y=%lf\n"sub(x,y);          /* function call*/
getch();
}
float add(a,b)          // function declaration – a,b formal arguments
float a,b;
{
return(a+b);
}
double sub(p,q)
double p,q;
{
return(p-q);
}
```

FUNCTION THAT RETURN MULTIPLE VALUES

✓ If a function uses Arguments to receive and send information to the calling function then it is said to be a function that return Multiple values. Here in this type function that return Multiple values Indirection operator (*) and Address Operator (&) are used.

```
#include<stdio.h>
void main()
{
        int a,b,c;
         float ratio(int a,int b,int c);
         int difference(int b,int c);
         printf("\n Enter the value of a,b and c:");
         scanf("%d%d%d",&a,&b,&c);
         printf("Ratio = %f\n",ratio(a,b,c));
         printf(" Difference = %d",difference(b,c);          getch();
}
float ratio(x,y,z)
int x,y,z;
{
        if(difference(y,z))
             return(x/y-z));
        else
        return(0,0);
}
difference(p,q)
{
        int p,q;
                if(p!=q)
                        return(1);
                else
        return(0);
}
```

RECURSION

- ✓ When a function is called repeatedly until specific condition is satisfied, then it is known as Recursion. A function is called recursive if a statement within the body of a function calls the same function.

Example:

```
#include<stdio.h>
#include<conio.h>
long fact(long);
void main()
{
long num, fac;
clrscr();
printf("Enter a number : ");
scanf("%ld",&sum);
fac=fact(num);
printf("The factorial of %ld is %ld",num,fac);          getch();
}
long fact(long x)
{
if(x<=1)
return(1);
else
return(x*fact(x-1));
}
```

STORAGE CLASSES

- ✓ Variables in C programs are totally different from other languages. We can use the same variable names in the C program in separate blocks. When we declare a variable it is available only to specific part or block of the program. Remaining block or other function cannot get access to the variable.
- ✓ The area or block of the C program from where the variable can be accessed is known as the scope of the variable. Scope of a variable is also defined as the region over which variable is visible or valid.
- ✓ The area or scope of the variable depends on its storage class, that is, where and how it is declared.
- ✓ The storage class of a variable tells the compiler
  - ▪ Storage area of the variable
  - ▪ Initail value of the variable if not initialized
  - ▪ Scope of the variable
  - ▪ Life of the variable that is how long the variable would be active in the program.
- ✓ There are four types of storage classes available in a C language. They are
  1. **Automatic Variables**
  2. **Static Variables**
  3. **External Variables**
  4. **Register Variables**

Automatic Variables

- ✓ Variables declared inside a block and local to block in which declared are said to be **Automatic** variables. These variables can be accessed by block in which they are declared. Another block cannot access its value.
- ✓ These variables created as new variable each time when function is called and destroyed automatically when the function is exited.
- ✓ Compiler treat variable declared inside block as automatic variable by default. Automatic variables are stored in memory. All variables declared inside the function is **auto** by default.
- ✓ Auto variables are safe that is, they cannot be accessed directly by other functions.

Example

```
        main()
        {
        int number;
        -----------;
        -----------;
        }
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
        main()
        {
        auto int number;
        -----------;
        -----------;
        }
```

- ✓ One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example1: Program to illustrate how automatic variables work

```
#include<stdio.h>
#include<conio.h>
void function1(void);
void function2(void);
main()
{
        int m=2000;
        function2();
        printf("%d\n",m);
}
```

```
void function1(void)
{
        int m=10;
        printf("%d\n",m);
}
void function2(void)
{
        int m=100;
        function1();
        printf("%d\n",m);
}
```

Output:

```
10
100
2000
```

Example2: Program to illustrate how automatic variables work

```
#include<stdio.h>
#include<conio.h>
void main()
        {
                clrscr();
                auto int x=5;
                {
                        auto int x=4;
                        {
                                auto int x=3;
                                printf("%d\t",x);
                        }
                  printf("%d\t",x);
                }
printf("%d\t",x);
getch();
}
```

Output:

3

4

5

External variables

- ✓ Variables that are active throughout the entire program are called as **external variables** (**global variables)**. External variables are declared outside the function body. This storage class is created when variable is declared global. No memory is reserved for the variable. Variable retain the value throughout the execution of a program. This storage class can be accessed by any function in same or different program file and change its value.
- ✓ For example , the external declaration of integer number and float length might appear as

```
int number;
float length=6.2;
main()
{
        -------;
        -------;
}
function1()
{
        -------;
        -------;
}
function2()
{
        ---------;
        ---------;
```

```
                    }
```

- ✓ The variables number and length are available for use in all the three functions.

Example1: Program to illustrate how External variables work

```
#include<stdio.h>
#include<conio.h>
int fun1(void);
int fun2(void);
int fun3(void);
int x;                  /*GLOBAL*/
main()
{
        x=10;           /*GLOBAL */
        printf("x=%d\n", x);
        printf("x=%d\n", fun1());
        printf("x=%d\n", fun2());
        printf("x=%d\n", fun3());
}
fun1(void)
{
        x=x+10;         /*GLOBAL*/
}
int fun2(void)
{
        int x ;         /*LOCAL*/
        x=1 ;
        return(x) ;
}
fun3(void)
{
        x=x+10;         /* GLOBAL*/
}
```

OUTPUT :

```
x=10
x=20
x=1
x=30
```

- ✓ Once a variable has been declared as global, any function can use it and change its value. Then subsequent functions can reference only that new value.
- ✓ One other aspect of a global variable is that is available only from the point of declaration to the end of the program. Consider the program segment shown below

```
main()
{
        y=5;
        --------;
        ---------;
}
int y;          /*Global Declaration*/
func1()
{
        y=y+1;
}
```

We have a problem here. As far as main is concerned, y is not defined. So the compiler will issue an error message.

External Declaration

- ✓ In the above program, the main cannot access the variable y as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class named as **“extern”**

```
main()
{
        extern int y;          /*External declaration*/
        --------;
        ---------;
}
func1()                          /*External declaration*/
{
        extern int y;
}
int y;                           /*Definition*/
```

Static Variables

- ✓ Static variable may be either an internal type or an external type depending on the place of declaration. Static variables declared within individual block. They are local to the block in which declared. It exists until the end of the program. Variable can be declared using the keyword **static**. Global and Local variable can be declared static. Static variables are initialized only once when they are compiled in a program.
- ✓ When the program is closed the function associated with that program is also excited and whenever it is visited again the same value exists. Internal static variable is declared inside a function. External static variable is declared outside a function. It is made available to all functions in a C program.

Example1: Program to illustrate how static variables work

```
#include<stdio.h>
#include<conio.h>
void start(void);
void main()
{
int i;
clrscr();
 for(i=1;i<3;i++)
 stat();
getch();
}
void stat(void)
```

```
{
static int x=0;
x=x+1;
printf("x=%d\n",x);
}
```

OUTPUT

```
x=1
x=2
x=3
```

Register Variables

- ✓ Register is a special storage area in a Central Processing Unit (CPU). There are 8 registers available inside a Computer. Register variable can be accessed only by block in which it is declared. It cannot be accessed by any other function.
- ✓ Register variable declared using keyword **register**. Both Local variable and formal parameter can be declared as a register. Register is used to increase the execution speed. Only integer or char variables are declared as register in most of the compilers but ANSI C supports all the data types.

Example: Program to illustrate how Register variables work

```
#include<stdio.h>
#include<conio.h>
void main()
{
        register int x;
        clrscr();
        for(x=1;x<=10;x++)
        printf("%d",x);
        getch();
}
```

STRINGS

- ✓ A string is a sequence of character that is treated as single data item.

- ✓ In C language the group of characters, digits and symbols enclosed within quotation marks are called as strings.
- ✓ Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include
  - o Reading and writing strings
  - o Combining strings together
  - o Copying one string to another
  - o Comparing strings for equality
  - o Extracting a portion of string

Declaration & Initialization of Strings

- ✓ C does not support strings as a data type. But it allows us to represent strings as character arrays.
- ✓ The general form of declaration of a string variable is

  char string_name[size];

  Size determines the number of characters in the string_name. Some examples are

  char city[10];

  char name[20];
- ✓ When the Compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore the size should be equal to the maximum number of characters in the string plus one.
- ✓ Character arrays are initialized when they are declared. C permits a character array to be initialized in either of the following two forms.

  char city [9] = " NEW YORK";

  char city [9] ={'N','E','W',' ','Y','O','R','K','\0'};

- ✓ C also permits us to initialize a character array without specifying the number of elements. In such cases the size of the array will be determined automatically, based on the number of elements initialized, For example the statement

  char string []={'I','N','D','I','A','\0};

  defines the string as 6 element array.

- ✓ We can also declare the size much larger than the string size in the initializer. For example the statement,

  char str[10]=”INDIA”;

  is permitted. In this case, the computer creates a character array of size 10, places the value “INDIA” in it, terminates with null character and initializes all other elements with the null character.

Reading strings from the terminal

- ✓ Scanf can be used with **%s** format specification to read in a string of characters
  For example:

  char address [10];
  scanf(%s”, address);
- ✓ The problem with the scanf function is that it terminates its input on the first white space it finds.
- ✓ We can also specify the field width using the form **%ws** in the scanf statement for reading a specified number of character from the input string.
  For Example

  scanf (“%ws”,name);

  Here two things may happen,
  1. The width **w** is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
  2. The width **w** is less that the number of character in the string. The excess character will be truncated and left un read

Consider the Following statements:

char name[10];
scanf (“%5s”, name);

Example1:

The input string “**HAI**” will be stored as

H	A	I	\0	?	?	?	?	?	?

0 1 2 3 4 5 6 7 8 9

Example 2:

The input string "**WELCOME**" will be stored as

W	E	L	C	O	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading a line of text

- ✓ Scanf with **%s or %ws** can read only strings without white spaces. That is they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the **edit set conversion code % […]** that can be used to read a line containing a variety of characters, including white spaces.
- ✓ For example consider the following program segment

```
char line[80];
scanf("%[^\n],line];
printf("%s",line);
```

will read a line of input from the keyboard and display the same on the screen

Using getchar and gets functions

- ✓ We already discussed how to read a single character from the terminal using the function **getchar**. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus an entire line of text can be read and stored in an array. The reading is terminated when the **newline** character('\n')is entered and the null character is inserted at the end of the string.
- ✓ The getchar function call takes the form

```
char ch;
ch=getchar();
```

- ✓ Another and more convenient method of reading a string of text containing white spaces is to use the library function **gets**. This is a simple function with one string parameter and called as

```
gets(str);
```

str is a string variable declared properly. It reads character into **str** through keyboard until a new-line character is encountered and then appends a null character to the string. It does not skip white spaces.

For example:

```
char line [80];
gets(line);
printf("%s", line);
```

reads a line of text from the keyboard and displays it on the screen.

Writing Strings to the screen

- ✓ Printf function with %s format to print strings to the screen. The format %s can be used to display an array of character that is terminated by the null character. For example the statement

  printf("%s", name);

  can be used to display the entire contents of the array name.
- ✓ We can also specify the precision with which the array is displayed. For example the specification **%10.4s** indicates that the first four characters are to be printed in a field width of 10 columns.

Example Program: Display the string under various format specification

```
#include<stdio.h>
#include<conio.h>
main()
{
        char state[15]="Andhra Pradesh";
        printf("\n\n");
        printf("--------------------------\n");
        printf("%15s\n", state);
        printf("%5s\n", state);
        printf("%15.6s\n", state);
        printf("%-15.6s\n", state);
        printf("%15.0s\n", state);
        printf("%.3s\n", state);
        printf("%s\n", state);
        printf("--------------------------\n");
}
```

OUTPUT

```
---------------------------
Andhra Pradesh
Andhra Pradesh
          Andhra
Andhra

And
Andhra Pradesh
---------------------------
```

Using Putchar and puts function

- ✓ Like getchar, C supports another character handling function putchar to output the values of character variables. It takes the following form

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

The function putchar requires one parameter. The statement is equal to **printf("%c",ch);**

- ✓ We can use this function repeatedly to output a string of character stored in an array as like the following example.

```
char name[6]="HAI"
for (i=0;i<5;i++)
putchar (name[i]);
putchar("\n");
```

- ✓ Another and more convenient way of printing string values is using the function **puts**

```
puts(str);
```

where **str** is a string variable containing a string value. This prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen. For example consider the following program segment

```
char line(80);
```

```
gets(line);
puts(line);
```

reads a line of text from the keyboard and displays it one screen. This syntax is very simple compared to using the scanf and printf functions

STRING LIBRARY FUNCTIONS

- ✓ C compiler supports a large number of string handling library functions that can be used to carry out many of the string manipulations

Functions	Description
Strlen()	Determines the length of a string
Strcpy()	Copies a string from source to destination
Strncpy()	Copies character of a string to another string upto specified length
Stricmp()	Compares characters of two strings.
Strcmp()	Compares two strings
Strncmp()	Compares characters of two strings upto the specified length
Strnicmp()	Compares characters of two strings upto the specified length, Ignores case
Strlwr()	Converts uppercase characters of a string to lowercase
Strupr()	Converts lowercase character of a string to uppercase
Strudp()	Duplicates a string
Strchr()	Determines the first occurrence of a given character in a string
Strrchr()	Determines the last occurrence of a given character in a string
Strstr()	Determines the first occurrence of a given string in another string
Strcat()	Appends source string to destination string
Strncat()	Appends source string to destination string upto the specified length
Strrev()	Reverses all character of a string
Strset()	Sets all character of a string with a given argument or symbol
Strnset()	Sets specified numbers of characters of a string with a given argument or symbol
Strspn()	Finds upto what length two strings are identical

Strpbrk()	Searches the first occurrence of the character in a given string and then displays the string starting from that character

1. strlen() Function

This function counts the number of character in a given string. The format of functions is

strlen(string);

Example Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
        char text[20];
        int len;
        clrscr();
        printf("Enter the text:\n");
        gets(text);
        len=strlen(text);
        printf("length of the string=%d",len);
}
```

OUTPUT

```
Enter the text
Welcome
Length of the string: 7
```

2. strcpy()function

This function almost like a string-assignment operator. this function copies the contents of one string to another. The format of strcpy() is

strcpy(string1, string2);

Example Program

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

```
#include<conio.h>
#include<string.h>
main()
{
        char original[20],duplicate[20];
        clrscr();
        printf("enter the string:");
        gets(original);
        strcpy(duplicate,original);
        printf("\n original string:%s".original);
        printf("\n  duplicate string:%s",duplicate);
        getch();
}
```

OUTPUT:

```
Enter the string:     SMVEC
Original string:      SMVEC
Duplicate String:     SMVEC
```

3. strncpy()function

This function performs the same task as strcpy(). The only difference between them is that the former function copies a specified length of character from source to destination string. Whereas this function copies the whole source to the destination string. The format of the function is

strncpy(destination, source, n);

Example Program

```
        #include<stdio.h>
        #include<conio.h>
        #include<string.h>
        void main
        {
                char str1[15],str2[15];
                int n;
                clrscr();
        printf("enter source string:");
        gets(str1);
        printf("enter Destination string:");
        gets(str2);
        printf("enter the number of character to replace in the destination :");
        scanf("%d",&n);
        strncpy(str2, str1, n);
        printf("source string:%s, str1);
        printf("destination string:%s", str2);
}
```

OUTPUT

```
Enter source string          : wonderful
Enter destination string     :  beautiful
Enter number of characters to replace in destination: 6
Source string                : wonderful
Destination string           : wonderful
```

4. stricmp() function

This function compares two strings. The character of the strings may be in lower case or upper case. The function does not discriminate between them, that is, this function compares two strings without case. If the strings are same it returns to zero otherwise a non-zero value.

Example Program

```
    #include<stdio.h>
    #include<conio.h>
    #include<string.h>
    void main
    {
        char str1[15],str2[15];
        int diff;
        clrscr();
        printf("enter string1:");
        gets(str1);
        printf("enter string2:");
        gets(str2);
        diff=stricmp(str1, str2);
        if(diff==0)
        puts("the two strings are identical");
        else
        puts("the two strings are not identical");
        getche();
    }
```

OUTPUT:

```
    Enter string1 :     WELCOME
    Enter string2 :     welcome
    The two strings are identical
```

5. strcmp() function

We can also use strcmp() function instead of stricmp(). The only difference between them is that the former function discriminates between small and capital letters.

6. strncmp() function

A comparison of two strings can be made upto certain specified length. The function used for this is **strncmp()**. This function is the same as **strcmp()** but it compares the character of the string to a specified length. The format of tehis function is as follows.

strncmp(source,target, argument)

Example Program

```
    #include<stdio.h>
    #include<conio.h>
    #include<string.h>
    void main
    {
        char str1[15],str2[15];
        int n, diff;
        clrscr();
```

```
                printf("enter string1:");
                gets(str1);
                printf("enter string2:");
                gets(str2);
                printf("\n Enter length up to which comparison is to be made");
                scanf("%d",&n);
                diff=strncmp(str1,str2,n);
                printf("the two strings are identical up to %d character",n);
                else
                puts("the two strings are differenct");
                getch();
        }
```

OUTPUT

```
Enter string1:                  WELCOME
Enter string2:                  WELLDONE
Enter length up to which comparison is to be made:      3
The two strings are identical up to 3 characters
```

7. strnicmp () function

We can also use **strnicmp()** function instead of **strncmp().** The only difference between them is that the **strncmp()** discriminates between small and capital letter whereas the **strnicmp()** does not. The output of the above program with **strnicmp()** in place of **strncmp()** will be as follows.

```
Enter string1:                  WELCOME
Enter string2:                  welldone
Enter length up to which comparison is to be made:      3
The two strings are identical up to 3 characters
```

8. strlwr & strupr() function

strlwr function an be used to convert any string to a lower case. When you are passing any upper case string to this function it converts it into lower case. **Strupr()** function can be used convert lower case to upper case.

The format of strlwr() function is **strlwr(string);**

The format of strupr() function is **strupr(string);**

Example Program(Upper case to lower case):

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char str1[15];
        clrscr();
        printf("enter string in upper case:");
        gets(str1);
        printf("after strlwr(): %s", strlwr(str1));
}
```

OUTPUT

```
Enter string in upper case: WELCOME
After strlwr            : welcome
```

9. strudp () function

This function is used for duplicating a given string at the allocated memory which is pointed by a pointer variable. The format of this function is as follows

text2=strudp(text1);

where text1 is a string

text2 is a pointer

Example Program: Entering the string and getting the duplicate

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char text1[15], *text2;
        clrscr();
        printf("enter text:");
        gets(text1);
        text2=strudp(text1);
        printf("Original string=%s\n Duplicate String:%s", text1,text2);
    }
```

OUPTUT

Enter text: Have a nice day

Original string: Have a nice day

Duplicate string: Have a nice day

10. strchr() Function

This function returns the pointer to the first occurrence of a given character in the string. The format of this function is as follows

chp=strchr(string,ch);

where **string** is a character array, **ch** is character variable, **chp** is a pointer which collects the address returned by **strchr()** function The format of this function is **strchr(string,character)**

Example Program/ Program to find the occurrence of a given character in the string

```
    #include<stdio.h>
    #include<conio.h>
    #include<string.h>
    void main
    {
        char string[30], ch, *chp;
        clrscr();
        printf("enter text:");
        gets(string);
        printf("\n character to find");
        ch=getchar();
        chp=strchr(string,ch);
        if(chp)
        printf("character %c found in string",ch);
```

```
		else
		printf("character %c not found instring",ch);
	}
```

OUTPUT

```
Enter  text: Have a nice day
Character to find:     n
Character n found in string
```

11. strrchr() Function

In place of **strchr()** one can use **strrchar().** The difference between them is that the **strchr()** searches for occurrence of character from the beginning of the string whereas **strrchr()** searches occurrence of character from the end.

12. strstr() Function

This function finds the second string in the first string. It returns the pointer location from where the second string starts in the first string. In case the first occurrence in the string is not observed, the function returns a NULL character. The format of this function is

strstr(string1,string2);

Example Program/ Program to find the occurrence of a second string in first string

```
	#include<stdio.h>
	#include<conio.h>
	#include<string.h>
	void main
	{
		char line1[35],line2[35], *chp;
		clrscr();
		puts("enter line1:");
		gets(line1);
		puts("enter line2:");
		gets(line2);
		chp=strstr(line1,line2);
		if(chp)
```

```
        printf("'%s'string is present in given string",line2);
        else
        printf("'%s' string is not present in given string",line2);
    }
```

OUTPUT

```
Enter line1     : Have a nice day
Enter lin2      : day
'day' string is present in the given string
```

13. strcat() Function

This function appends the target string to the source string. Concatenation of two strings can be done using this function. The format of this function is as follows

strcat(tex1,text2);

Example Program: Joining two strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char text1[30], text2[30];
        puts("enter text1");
        gets(text1);
        puts("enter text2");
        puts(text2);
        strcat(text1," ");
        strcat(text1, text2);
        clrscr();
        printf("%s\n", text1);
}
```

OUTPUT

```
Enter text1    :        Have a
Enter text2     :          nice day
Have a nice day
```

14. strncat() Function

This function is the same as that of strcat(). The difference between them is that the former does the concatenation of the strings with another up to a specified length. The format of this function is

strncat(text1,text2,n);

Example Program: Joining two strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char text1[30], text2[30],n;
        puts("enter text1");
        gets(text1);
        puts("enter text2");
        puts(text2);
        printf("enter number of characters to add:");
        gets(n);
        strcat(text1," ");
        strncat(text1, text2);
        clrscr();
        printf("%s\n", text1);
}
```

OUTPUT

```
Enter text1     :          Have a
Enter text2     :          nice day
Enter number of characters to add: 4
Have a nice
```

15. strrev() Function

This function simply reverses the given string. The format of this function is

strrev(sring);

Example Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char text[15];
        clrscr();
        puts(“enter string1:”);
        gets(text);
        puts(“reverse string”);
        puts(strrev(text));
}
```

OUTPUT

```
Enter string
Welcome
Reverse string
emoclew
```

16. strset() Function

This function replaces every character of a string with the symbol given by the programmer, that is the elements of the strings are replaced with the arguments given by the programmer. The format of this function is

strset(string, symbol)

Example Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
```

```
{
        char text[15];
        char symbol;
        clrscr();
        puts(“enter string:”);
        gets(text);
        puts(“enter symbol for replacement:”);
        scanf(“%c”, &symbol);
        printf(“Before strset():%s\n”, text);
        strset(string, symbol);
        printf(“After strset(); %s\n”, text);
}
```

OUTPUT

```
Enter string   :        Computer
Enter symbol for replacement:      x
Before strset():        Computer
After strset():         XXXXXXX
```

17. strnset() Function

This function is the same as that of **strset().** Here the specified length is provided. The format of this function is

strnset(string, symbol, n);

where n is the number of characters to be replaced.

Example Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main
{
        char text[15];
        char symbol;
        int n;
        clrscr();
        puts(“enter string:”);
        gets(text);
        puts("enter symbol for replacement:");
        scanf("%c", &symbol);
        puts("how many string character to be replaced");
        scanf("%d",&n);
        printf("Before strset():%s\n", text);
        strset(string, symbol,n);
        printf("After strset(); %s\n", text);
    }
```

OUTPUT

```
Enter string   :          Computer
Enter symbol for replacement:       x
How many string characters to be replaced: 4
Before strset():          Computer
After strset():           XXXXuter
```

18. strspn() Function

This function returns the position of the string from where the source array does not match with target one. The format of this function is

strspn(string1, string2);

Example Program: Indicate after what character the lengths of the two strings have no match

```
    #include<stdio.h>
    #include<conio.h>
    #include<string.h>
    void main
    {
        char text1[30], text2[30];
        int length;
        clrscr();
        puts("enter text1");
        gets(text1);
        puts("enter text2");
        gets(text2);
        length= strspn(text1,text2);
        printf("After %d character there is no match\n", length);
    }
```

OUTPUT

```
Enter text1   :     GOOD MORNING
Enter text2   :     GOOD BYE
After 5 characters there is no match
```

19. strpbrk() Function

This function searches the first occurrence of a character in the given string and then it displays the string starting from that character. This function returns the pointer position to the first occurrence of the character **text2[2**] string in the string **text1[20].** The format of this function is

strpbrk(text1,text2)

Example Program:

```
    #include<stdio.h>
    #include<conio.h>
    #include<string.h>
    void main
    {
        char *ptr;
        char text1[20], text2[2];
        clrscr();
        puts("enter text1");
        gets(text1);
        puts("enter character");
        gets(text2);
        ptr=strpbrk(text1,text2);
        puts("string from given character");
        printf(ptr);
    }
```

OUTPUT

```
Enter text1     :         Have a nice day
Enter character:         n
string from given character: nice day
```