Veer Narmad South Gujarat University, Surat

C.K.Pithawalla College of Commerce-Management-Computer Application

Semester - I

104–Computer Programming and Programming Methodology (CPPM)

Unit 1: Introduction

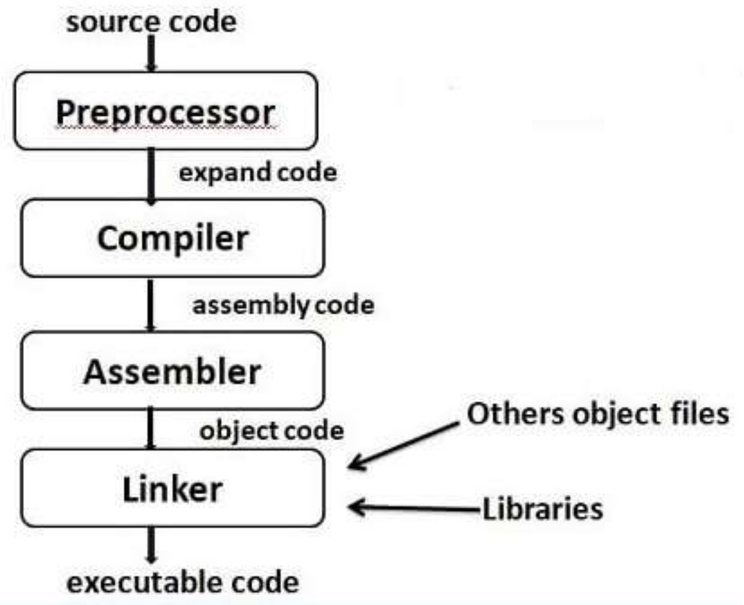
1.1. Concepts of Programming Language:

1.1.1. Introduction to Source Code, Object Code and Executable Code

Source Code:

Source code refers to high level code or assembly code which is generated by human/programmer.

- Source code is easy to read and modify.
- It is written by programmer by using any High Level Language or Intermediate language which is human-readable.
- Source code contains comments that programmer puts for better understanding.
- Source code is provided to language translator which converts it into machine understandable code which is called machine code or object code.
- Computer can not understand direct source code, computer understands machine code and executes it.
- In simple we can say source code is a set of instructions/commands and statements which is written by a programmer by using a computer programming language like C, C++, Java, Python, Assembly language etc. So statements written in any programming language is termed as source code.



Object Code :

- Object code refers to low level code which is understandable by machine.
- Object code is generated from source code after going through compiler or other translator.
- It is in executable machine code format.
- Object code contains a sequence of machine understandable instructions to which Central Processing Unit understands and executes.
- Object file contains object code.
- It is considered as one more of machine code.
- It is the output of a compiler or other translator.

- We can understand source code but we cannot understand object code as it is not in plain text like source code rather it is in binary formats.

Executable Code

- Executable code is a file or a program that indicates tasks according to encoded instructions the CPU can directly execute.
- A machine code file can be immediately *executable* (i.e., runnable as a program), or it might require *linking* with other object code files (e.g. *libraries*) to produce a complete executable program.

1.1.2. Algorithm and Flowchart

Algorithm

- The step-by-step procedure to solve the problem is called Algorithm. These steps also provide the way to solve the problem. It is also known as logic and usually written in simple English language.
- An Algorithm may be defined as:
  - A **well-defined** computational procedure that transforms inputs into outputs, for achieving the **desired** results.
  - A set of instructions for solving a problem.
- An algorithm written for the first time needs reviewing to:
  - Determine the correctness of various steps
  - Reduce the number of steps if possible
  - Increase the speed of solving the problem

- **Features of Algorithms:**
  - It should have fixed number of steps.
  - Steps are to be simple.
  - Inputs and outputs are to be clearly defined.
  - Steps should be correct and accurate.

E.g.: Write an algorithm to add two numbers.

Step-1. Start
Step-2. Get the two numbers
Step-3. Add the two numbers
Step-4. Print the sum
Step-5. Stop

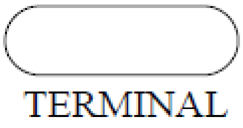
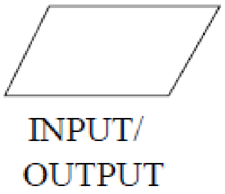
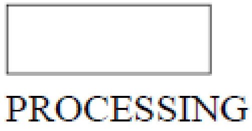
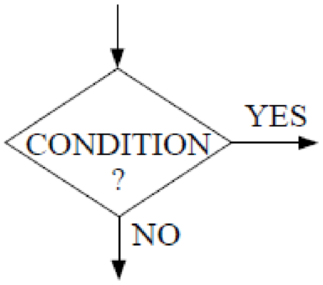
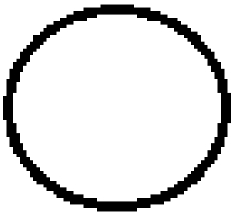
E.g.: Write an algorithm to print whether a number is positive or negative.

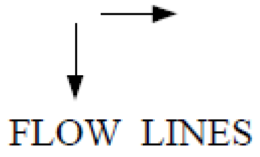
Step-1. Start
Step-2. Accept a number from the user
Step-3. Check whether the number is greater than 0 if so then display "Positive" and go to step – 5 else go to step 4
Step-4. Display "Negative"
Step-5. Stop

Flowchart

- A flowchart is a graphical representation of the sequence of operations in an information system or program.
- Program flowcharts show the sequence of instructions in a single program.
- Flowchart uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed.
- Since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected easily.
- Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.

Flowchart Symbols

	1. **Terminal Box:** The terminal symbol, as the name implies, is used to indicate the starting (BEGIN), stopping (END), and pause (HALT) in the program logic flow. It is the first symbol and the last symbol in the program logic.
	2. **Input/ Output Box:** The input/output symbol is used to denote any function of an input/output device in the program. If there is a program instruction to input data from an input device or we want to output the results to any output device, are indicated in the flowchart with an input/output symbol.
	3. **Processing Box:** A processing symbol is used in a flowchart to represent arithmetic and data movement instructions. Thus, all arithmetic processes such as adding, subtracting, multiplying and dividing are shown by a processing symbol.
	4. **Decision Box:** The decision symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternative points is possible. During execution, the appropriate path is followed depending upon the result of the decision.
	5. **Connector: T**here are instances when a flowchart becomes too long to fit in a single page and the use of flow lines becomes impossible. Thus, whenever a flowchart spreads over more than one page, it is useful to utilize the connector symbol as a substitute for flow lines. This symbol represents an entry from, or an exit to another part of the flowchart. A connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link.

	6. **Flow Lines:** Flow-lines with arrowheads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right.

Rules for Flowcharting:

1. First formulate the main line of logic, and then incorporate the details.
2. Maintain a consistent level of detail for a given flowchart.
3. Do not give every detail on the flowchart. A reader who is interested in greater details can refer to the program itself.
4. Words in the flowchart symbols should be common statements and easy to understand.
5. Be consistent in using names and variables in the flowchart.
6. Go from left to right and top to bottom in constructing the flowchart.
7. Keep the flowchart as simple as possible. The crossing of flow lines should be avoided as far as possible. Flow-lines must be in one-direction only.
8. Properly labelled connectors should be used to link the portions of the flowchart on different pages.

Advantages of Flowcharting:

1. Since a flowchart is a pictorial representation of a program, it is easier for a programmer to understand and explain the logic of the program to some other programmer.
2. It analyses the problem effectively.
3. Flow-chart is a road-map to efficient coding.
4. It assists in reviewing and debugging of a program.
5. It provides effective program documentation.

Disadvantages of Flowcharting:

1. Flowcharts are very time-consuming and laborious to draw.
2. Redrawing a flowchart, if there are any changes, is tedious.
3. There are no standards determining the amount of detail that should be included in a flowchart.

Flowchart supports the following logic structures:

1. **Sequencing:** This logical structure is used for performing instructions one after another in sequence. Hence, for this logic instructions are written in the order, or sequence, in which they are to be performed. The flow is from top to bottom.
2. **Branching / Selection:** It is non-sequential logic which is also known as Decision logic. Statement may or may not be executed based on some condition. The condition is written inside the decision box in the flow chart. If the condition is true a particular path is followed otherwise another path is executed.
3. **Iteration/ Looping:** When one or more conditions may be executed several times, depending on some condition, we use looping. We can use the same decision box to check the execution times i.e. the number of times the statements are executed. Loops can be fixed or variable.

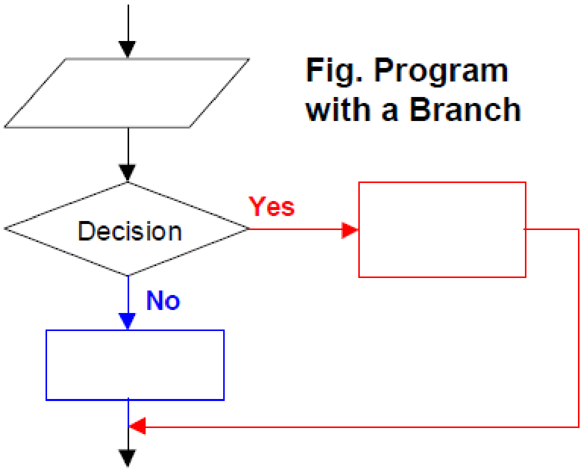


Fig. Program with a Branch

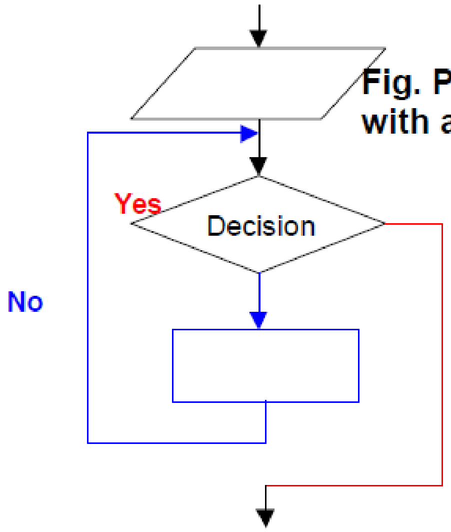


Fig. Program with a Loop

1.1.3. Concept of Structured Programming Language

Programming

- Programming is a process that starts with a problem that a program is to solve.
- The first step in the process is for the programmer to thoroughly understand that problem --- to understand what information will be given to the program or what information the program needs (this is called the input) and what answers (and in what format) the program should produce (this is the output).
- The programmer then proceeds to design an algorithm to solve the problem and translates that algorithm into a programming language.
- At this point the programmer types the program into the computer and tests it to see if it works correctly.

Two phases involved in the design of any program:

Problem Solving Phase: In the problem-solving phase the following steps are carried out:

i. Define the problem
ii. Outline the solution
iii. Develop the outline into an algorithm
iv. Test the algorithm for correctness

Implementation Phase: The implementation phase comprises the following steps:

v. Code the algorithm using a specific programming language
vi. Run the program on the computer
vii. Document and maintain the program

Types of Programming Languages:

Computer cannot do anything on its own. It needs programs which are group of instructions, which when executed solves the problem. For this purpose, different languages are developed for performing different types of work on the computer. Basically, languages are divided into two categories according to their interpretation.

1. Low Level Language (Machine Language)
2. Assembly Language
3. High Level Languages

1. Low Level Language (Machine Language):

- Low level computer languages are machine codes or close to it.
- Computer can't understand instructions given in high level languages or in English. It can only understand and execute instructions given in the form of machine language i.e. language of 0 and 1. Machine Language is basically the only language which computer can understand.
- Machine language require detailed knowledge of every aspect of the computer's architecture, including processor register function, I/O details, memory layout and access methods, etc.
- Machine language is sometimes also referred as the binary language because it has base-2 i-e, the language of 0 and 1.
- Very few computer programs are actually written in machine language.

Advantages:

1. Faster execution **2.** Computer easily understands

Disadvantages:

1. Because it consists of only 0's and 1's it is very difficult to code and modify.
2. It is machine dependent and not portable. **3** Error solving is also difficult.

2. Assembly Level Language:

- It is an intermediate language which is written using mnemonics and 1's and 0's.
- It is a 2nd generation language.
- Since the computer cannot understand it directly, it needs to be translated into machine-level language.
- It requires a program called **Assembler** for converting assembly code to machine code.

Advantages:

1. Easier to program as compared to machine level language.
2. Easy to understand
3. Easy to locate and correct errors

Disadvantages:

1. It is machine dependant and so not portable.
2. Takes more execution time than machine level language.
3. Compute cannot understand directly it requires translator (Assembler).

3. High Level Language:

- High level language programs are written in English language.
- The programs cannot be executed directly. It needs to be converted to machine level language for execution.
- It requires a translator called '**Compiler**' or '**Interpreter**' for converting high-level to machine-level language.
- E.g.: BASIC, FORTRAN, PASCAL, C, COBOL, etc. They are also known as Procedure-oriented languages.
- Programmer needs to specify complete instructions inorder to perform a task.

Advantages:

1. Easier to program.
2. Easy to write and understand.
3. Portable i.e. Machine independent.
4. Easy to locate and correct errors

Disadvantages:

1. Takes more execution time.
2. Compute cannot understand directly it requires translator (Compiler/Interpreter).

Structured Programming

- Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.
- Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections.
    - A defined function or set of similar functions is coded in a separate module or sub-module, which means that code, can be loaded into memory more efficiently and that modules can be reused in other programs.
    - After a module has been tested individually, it is then integrated with other modules into the overall program structure.
    - The reason for the evolution to structured programming is the need for well-organized programs that are:
        (a) Easier to Design
        (b) Easy to Read and understand
        (c) Easy to Modify
        (d) Easy to Test and Debug
        (e) Combine with other programs
        (f) More Reliable
    - Examples of Structured Programming Languages: C, Ada and Pascal.
    - **The Structure Theorem:** It is possible to write any computer program by using only three (3) basic control structures, namely:
        1. Sequential
        2. Selection (if-then-else)
        3. Repetition (looping, DoWhile)

1. **Sequencing:** This logical structure is used for performing instructions one after another in sequence. Hence, for this logic instructions are written in the order, or sequence, in which they are to be performed. The flow is from top to bottom.
2. **Branching / Selection:** It is non-sequential logic which is also known as Decision logic. Statement may or may not be executed based on some condition. The condition is written inside the decision box in the flow chart. If the condition is true a particular path is followed otherwise another path is executed.
3. **Iteration/ Looping:** When one or more conditions may be executed several times, depending on some condition, we use looping. We can use the same decision box to check the execution times i.e. the number of times the statements are executed. Loops can be fixed or variable.

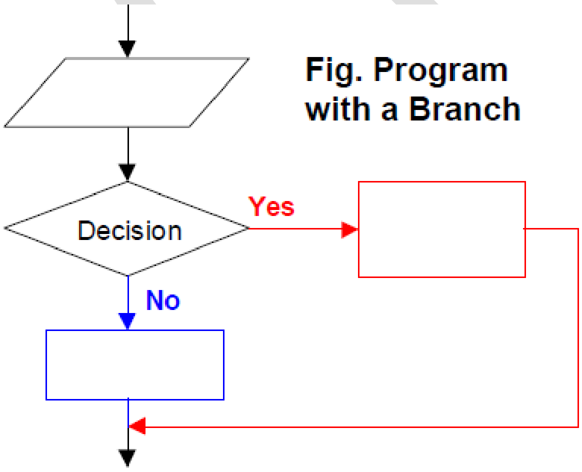


Fig. Program with a Branch

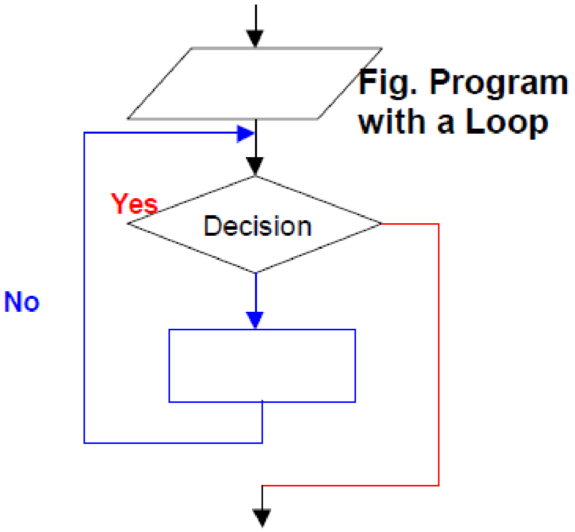


Fig. Program with a Loop

1.2. Concepts of Editor, Compiler and Interpreter:

Compiler

- A program which translates a high-level language program into a machine language program is called a **compiler**.
- It checks all kinds of limits, ranges, errors etc. But its program execution time is more, and occupies a larger part of the memory.
- It has low speed and low efficiency of memory utilization.

Interpreter

- An interpreter is a program which translates one statement of a high-level language program into machine codes and executes it.
- In this way it proceeds further till all the statements of the program are translated and executed.
- A compiler is nearly 5 to 25 times faster than an interpreter. An interpreter is a smaller program as compared to the compiler. It occupies less memory space. It can be used in a smaller system which has limited memory space.

Compiler	Interpreter
Scans the entire program first and then translates the whole program.	Scans the program line by line and translates immediately.
List of errors which are generated for the whole program are to removed and then the program is executed.	Syntax errors for each line are to be solved and then translated. Once all the lines are checked and translated, the program is executed.
Changes in program needs recompilation hence it takes more time.	It is quicker and easier to make changes in the program and thus development time can be reduced.
They use more memory space.	They use less memory space and reliable for smaller systems.

Editor

- In the field of programming, the term **editor** usually refers to source code editors that include many special features for writing and editing code.
- Notepad, Wordpad are some of the common editors used on Windows OS and vi, emacs, Jed, pico are the editors on UNIX OS.
- Features normally associated with text editors are — moving the cursor, deleting, replacing, pasting, finding, finding and replacing, saving etc.
- There are generally five types of editors as described below:

1. **Line editor:** Here we can only edit one line at a time or an integral number of lines. We cannot have a free-flowing sequence of characters. It will take care of only one line. Ex : Teleprinter, edlin, teco

2. **Stream editors:** In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means here we can type paragraphs. Ex : Sed editor in UNIX

3. **Screen editors:** In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer. Ex : vi, emacs, Notepad

4. **Structure Editor:** Structure editor focuses on programming languages. It provides features to write and edit source code. Ex : Netbeans IDE, gEdit.

5. **Text Editor:** A text editor is a software program that creates and manages text files. Ex. Notepad.

Types of Program Errors

- Once a program has been typed in, different types of errors may show up. These include:
  (a) Syntax/semantic errors **(c) Runtime errors**
  (b) Logic errors

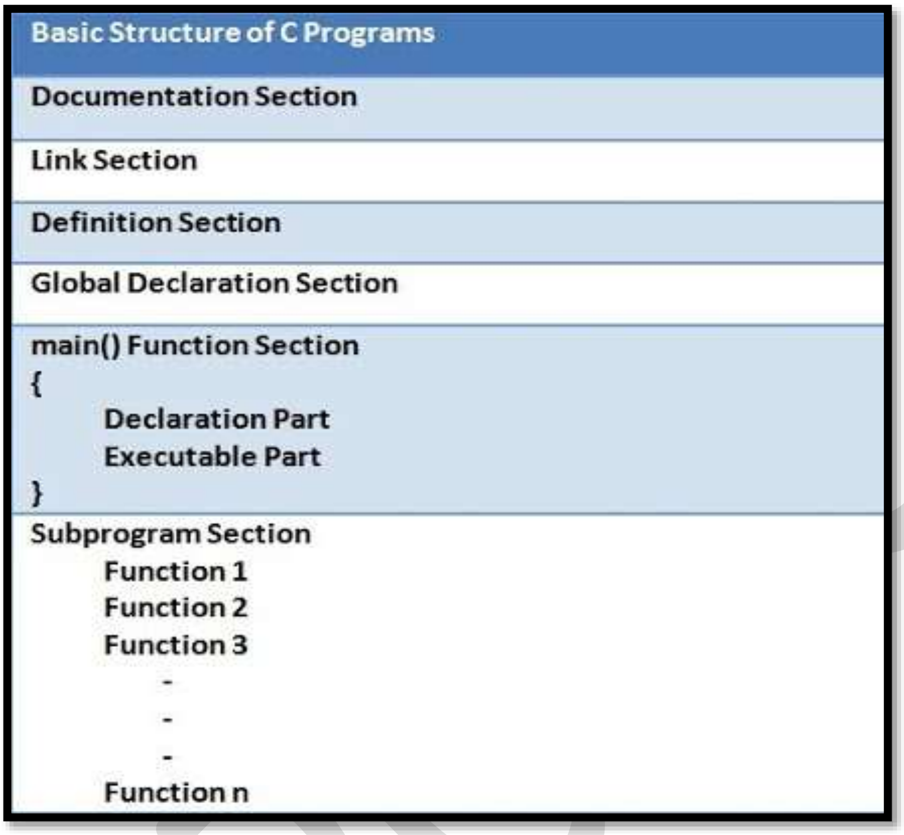
1. **Syntax/ Semantic Errors:**Syntax is a set of rules governing the structure of and relationship between symbols, words and phrases in a language statement. A syntax error occurs when a program cannot understand the command that has been entered.

2. **Logic Errors:** Logic refers to a sequence of operations performed by a software or hardware. Software logic or program logic is the sequence of instructions in a program. Logic errors are the errors that have been entered in the instructions created because of the mistake made by a programmer.

3. **Runtime Errors:** Runtime errors occur when a program is run on the computer and the results are not achieved due to some misinterpretation of a particular instruction. This could be something like dividing a number by zero which results in a very large value of quotient. It may also be any other instruction which a computer is not able to understand.

1.2.1. Basic Structure of a C Program

C program is divided into different sections. There are six main sections to a basic c program.



Basic Structure of C Programs
Documentation Section
Link Section
Definition Section
Global Declaration Section

main() Function Section
{
Declaration Part
Executable Part
}

Subprogram Section Function 1 Function 2 Function 3 - - - Function n

- **Documentation Section:**
  The documentation Section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer may use later. It gives anyone reading the code the overview of the code.

- **Link Section:**
  The Link section provides instructions to the compiler to link functions from the system library. It is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.

- **Definition Section:**
  It defines all the symbolic constants to be used in the program.

- **Global Declaration Section**
  This part of the code is the part where the global variables are declared. All the global variables used are declared in this part. **Global Variables** can be used in more than one function. The user-defined functions are also declared in this section.

- **Main Function**
  Every C-programs must have one **main()** function. Each main function contains 2 parts. A **declaration section** and an **Execution section**.

The declaration section is where all the variables are declared to be used in the execution section.

The execution section begins with the curly brackets and ends with the curly close bracket. These two sections must appear between the opening and closing curly braces. The closing brace of **main()** is the logical end of the program. There should be atleast one statement in this section. The statements in this section must end with a semi-colon **(;).**

- **Sub Programs Section:**
  This section contains all the UDFs (User Defined Functions) that are called in the main(). UDFs are generally placed immediately after the main().

```
// Program to calculate area of a circle           // Documentation Section
/* Name: Abc                                       /* is used for multi-line comments
   Title : Developer                               // is used for single line comments
    Proj: Xyz
*/
#include<stdio.h>                                  // Link Section
#include<conio.h>

#define PI  3.14                                   // Definition Section (Symbolic constant)

int r = 5;                                         // Global Declaration Section

void main()                                        // Start of Main function
{
        float area;                                // Declaration Section
        area = PI * r * r;                         // Executable section
        printf("\n Area = %f", area);
        displayline();
getch();
}                                                  // End of Main Function

void displayline()                                 //Sub Program Section
{
        printf("=======================");
}
```

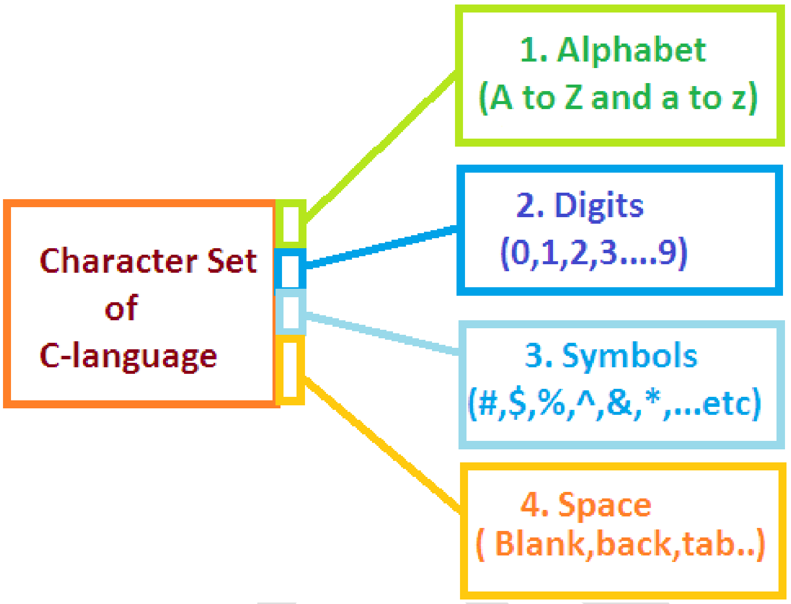
All the sections other than the main() may or may not be present.

1.2.2. Character-set, concept of variables and constants:

Character-set

- A program is a set of instructions that, when executed, generate an output. The data that is processed by a program consists of various characters and symbols. The output generated is also a combination of characters and symbols.

- As every language contains a set of characters used to construct words, statements, etc.,C language also has a set of characters which include alphabets, digits, and special symbols.
- C language supports a total of 256 characters.
- C language character set contains the following set of characters...
  1. **1.** Alphabets (Lower-case a- z and Upper-case A-Z)
  2. **2.** Digits (Digits from 0 – 9)



3. Special Symbols

, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
; (semi-colon)	[(left bracket)
: (colon)	] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)	) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)
|(vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
$ (dollar sign)	< (less than or opening angle bracket)
% (percentage sign)	# (hash sign)

4. White-Space Characters

a. Blank Space

b. Horizontal Tab

c. Carriage Return

d. New Line

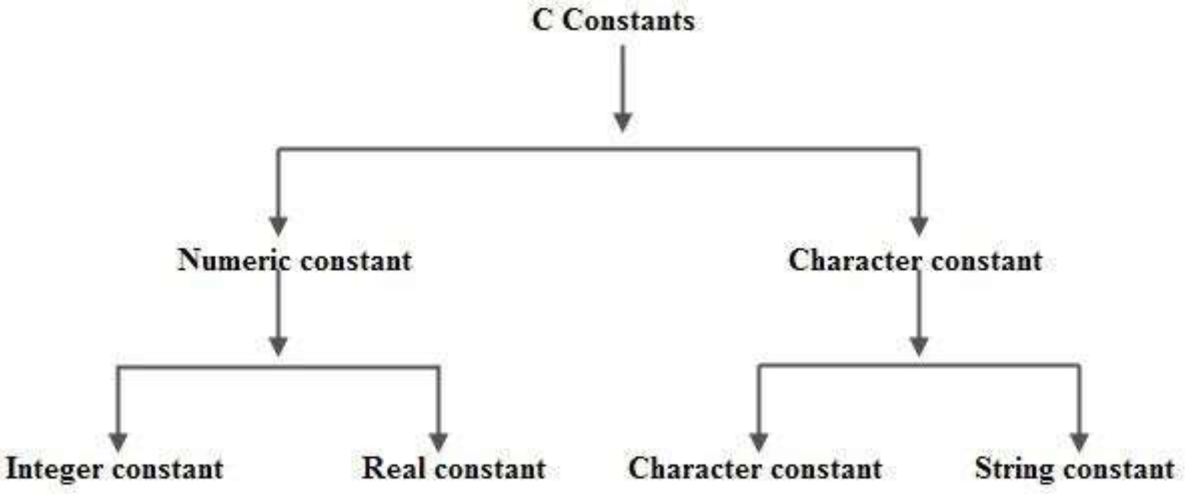
Trigraph Characters

- The symbols [ ] { } ^ \ | ~ # are frequently used in C programs, but in the late 1980s, these were not available on standard keyboards.
- To solve this problem, the C standard suggested the use of combinations of three characters to produce a single character called a trigraph.
- A **trigraph** is a **three-character sequence** that represents a single character.
- It means that **trigraph sequences** are the set of three characters starting from double question marks (??).

Trigraph	Equivalent
??=	# Number Sign
??/	\ BackSlash
??'	^ Caret
??(	[Left Bracket
??)	] Right Bracket
??!	|Vertical Bar
??<	{ Left Brace
??>	} Right Brace
??-	~ Tilde

Constants

- Constants are the fixed values that never change during the execution of a program.
- They are also called **Literals in C.**
- Constants can be classified into:
  - **Numeric Constants**
  - **Character Constants**



Numeric Constants:

1. **Integer constants**: An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

a. **Decimal constant** contains digits from 0-9 such as, Example, 111, 1234
b. **Octal constant** contains digits from 0-7, and these types of constants are always preceded by 0. Example, 012, 065
c. **Hexadecimal constant** contains a digit from 0-9 as well as characters from A-F. Hexadecimal constants are always preceded by 0X. Example, 0X2, 0Xbcd

2. **Real constants**: Real Constants consists of a fractional part in their representation. They are represented by numbers containing fractional parts like 26.082. Example of real constants are 1.52, 25.703, etc.

- Real Numbers can also be represented by exponential notation. The general form for exponential notation is **mantissa e exponent**. The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. Example : 6.3 E+02 7.2 E-02
- Real constants are also called **Floating-point constants.**

Character Constants:

1. **Single Character constants:** A Single Character constant represents a single character which is enclosed in a pair of single quotes (‘ ‘). Example : ‘A’ , ‘c’, ‘F’, etc. All character constants have an equivalent integer value which are called ASCII Values.

2. **String Constants:** A string constant is a set of characters enclosed in double quotation marks (“ “). The characters in a string constant sequence may be a alphabet, number, special character and blank space. Example of string constants are: “Hello”, “A123”, etc.

3. **Backslash Character Constants [Escape Sequences]**:
   - Certain ASCII characters are unprintable, which means they are not displayed on the screen or printer.
   - These characters perform other functions aside from displaying text. Examples are backspacing, moving to a newline, or ringing a bell.
   - They are used in output statements and also known as Escape Sequences.
   - An Escape sequence usually consists of a backslash and a letter or a combination of digits.
   - An escape sequence is considered as a single character but a valid character constant.
   - They start with a backslash and are enclosed in single quotes.

   Given below is the table of escape sequence and their meanings

Constant	Meaning	Constant	Meaning
‘\a’	Audible Alert (Bell)	‘\r’	Carriage Return
‘\b’	Backspace	‘\t’	Horizontal tab
‘\f’	Form feed	‘\v’	Vertical Tab
‘\n’	New Line	‘\”	Single Quote
‘\”‘	Double Quote	‘\\’	Back Slash
‘\?’	Question Mark	‘\0’	Null

Variables

- A variable is an identifier which is used to store some value.
- Constants can never change at the time of execution.
- Variables can change during the execution of a program and update the value stored inside it.
- A single variable can be used at multiple locations in a program.
- A variable name must be meaningful and should represent the purpose of the variable.
- A variable must be declared first before it is used somewhere inside the program.
- A variable name is formed using characters, digits and an underscore.

Rules for declaring a variable:

1. A variable name should consist of only characters, digits and an underscore.
2. A variable name should not begin with a number.
3. A variable name should not consist of whitespace.
4. A variable name should not consist of a keyword.
5. 'C' is a case sensitive language that means a variable named 'age' and 'AGE' are different.

Examples of valid variables are: age, a, b, gross_salary, n1, n2, etc.
1n, 2n, _a, abc@123, etc. are invalid variable names.

Declaration of variable:

- After giving suitable names to a variable, we declare them for the compiler.
- Declaration a variable does two things:
    - It tells the compiler the name of the variable.
    - It also specifies what type of data will be hold by the variable. (Data Type)
- In C, declaration of a variable is compulsory before it is used in the program. We declare all the variables at the beginning of the main function.

  Syntax : **datatype** **var1, var2, ... , varn;**
- Variables are separated by commas and semi-colon is put at the end.

  Example:
  ```
  int a, b;
  float  salary;
  char ch;
  ```

1.2.3. Identifiers, Literals and Keywords (C Tokens)

C Tokens

- **TOKEN** is the smallest unit in a 'C' program.
- It is each and every word and punctuation that we come across in our C program.
- The compiler breaks a program into the smallest possible units (tokens) and proceeds to the various stages of the compilation.
- A token is divided into six different types:
    - **Keywords**
    - **Operators**
    - **Strings**
    - **Constants/ Literals**
    - **Special Characters**
    - **Identifiers.**

Keywords:

- In 'C' every word can be either a keyword or an identifier.
- Keywords have fixed meanings, and the meaning cannot be changed.
- They act as a building block of a 'C' program.
- There are a total of 32 keywords in 'C'.
- Keywords are written in lowercase letters.

Following are the keywords available in C.

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

Literals/ Constants: Explained Above in detail.

Identifiers:

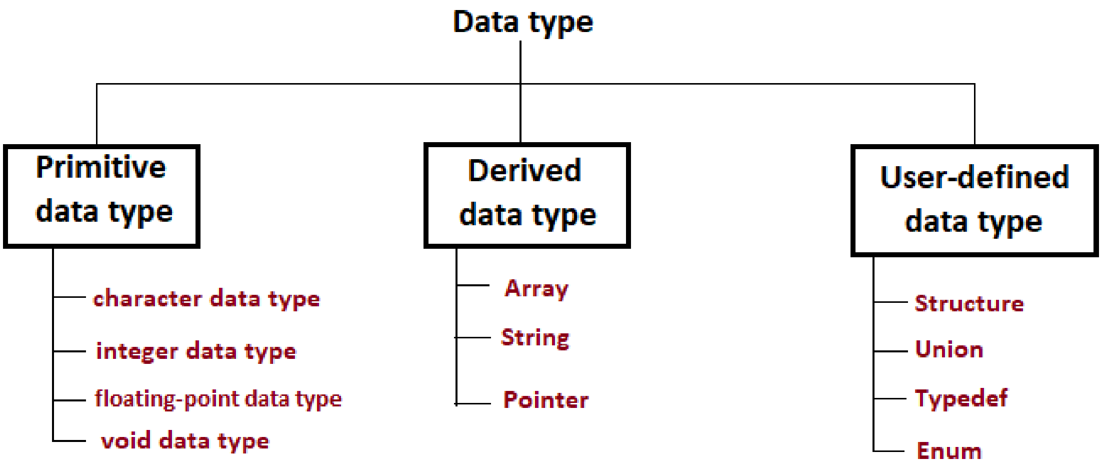
- An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc.
- Identifiers are the user-defined names consisting of 'C' standard character set.
- As the name says, identifiers are used to identify a particular element in a program.
- Each identifier must have a unique name.

Following rules must be followed for identifiers:

1. The first character must always be an alphabet or an underscore.
2. It should be formed using only letters, numbers, or underscore.
3. A keyword cannot be used as an identifier.
4. It should not contain any whitespace character.
5. The name must be meaningful. Only 1st 31 characters are significant.

1.2.4. Datatypes in C

- 'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application.
- ANSI C supports the following three classes of data types:
  - Primary or Fundamental or Basic Datatypes
  - Derived Datatypes
  - User-Defined Datatypes



Primary Data types

- All C Compilers support the five fundamental datatypes namely:

int	char	float	double	void

1. int type:

- Integers datatypes are whole numbers that can have both zero, positive and negative values but no decimal values. Example: int a = 12 // valid but int a = 12.5 // invalid
- The range for an integer data type varies from machine to machine.
- The standard range for an integer data type is -32768 to 32767 if the machine is 16-bits (2 bytes).
- We can also have integers of **signed** and **unsigned** type.
- An **unsigned** variable type of int can hold zero and positive numbers, and a **signed** int holds negative, zero and positive numbers.
- Therefore, 2 byte unsigned int numbers can store numbers from 0 to 65,536 and signed int numbers from -32,768 to 32,767.
- The keyword **int** is used to declare integer data type. If integer data is small then we can use keyword **short** or **short int** and to store long integer number we can use **long** or **long int** keyword.

- We can't store decimal values using int data type.
- If we use int data type to store decimal values, decimal values will be truncated and we will get only whole number.

2. char type:

- A single character can be defined by **char** type data.
- Characters are stored in 8-bits (1 Byte) and so the size of char data type is 1 byte.
  Example : char test = 'h';
- It may be observed that small **int** value can be stored in **char** variables and **char** values can be stored in **int** variables.

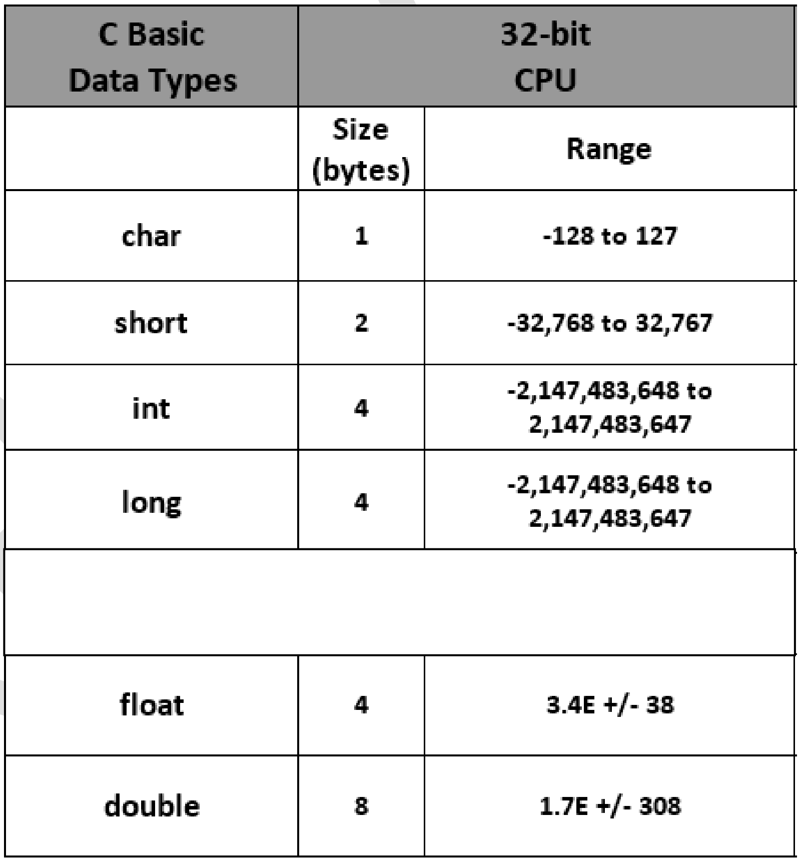
- This is because **char** type is classified under the integral data type as the storage occurs in the form of ASCII values which are used to represent every character.
- Therefore, for every single character on the keyboard there is an ASCII value (which is an int). **Example:** ASCII value of A is 65 and a is 97.
- So if we print 'A' as int we get 65 and if we print 65 as char we get 'A'.
- We can store only single character using char type variable. To store entire line of text we use character arrays.

3. **float type:**

- Floating point (Real) numbers are stored in 32 bits (4 Byte) with 6 digits of precision. Therefore, the size of float type is 4 bytes. **Example:** float f = 1.234567
- The range of float type is 3.4 E – 38 to 3.4 E + 38.

4. **double type:**

- A float type cannot hold a very very large number and also numbers with precision of more than 6 digits.
- Therefore, **double** type is used to provide more accuracy.
- **Double** is stored in 64 bits i.e the size of doube type is 8 bytes.
- It has 14 digits of precision.
- The range of double type is 1.7 E – 308 to 1.7 E + 308
- We can double the size of **double** type by specifying **long** keyword before double.
- **Long double** uses 10 bytes of storage.



C Basic Data Types	32-bit CPU	
	Size (bytes)	Range
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E +/- 38
double	8	1.7E +/- 308

5. **void type:**

- The **void** type specify that it has no value.
- We generally use **void** keyword to specify return type of a function.
- When a function does not return any value, we specify the return type as void which means nothing. **Example: void** main()
- Void type can also be used to represent generic types.

1.2.5. Source Code, Executable Code and Object code – Discussed earlier

Unit 2: I/O Statements and Operators

2.1 Input Output Statements

2.1.1 Concepts of Header Files (STDIO, CONIO):

- **C language** has many libraries that include predefined functions to make programming easier.
- In C language, header files contain the set of predefined standard library functions.
- We request to use a header file in our program by including it with the C preprocessing directive **"#include"**.
- A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source code files.
- There are two types of header files:
  - **Pre-existing header files:** Files which are already available in C/C++ compiler we just need to import them. Eg. : #include<stdio.h>, #include<conio.h>
  - **User-defined header files:** These files are defined by the user and can be imported using "#include".

Some of the pre-exsisting Header files

- **<stdio.h>:** The header file stdio.h stands for Standard Input Output. It has the information related to input/output functions.
- **<conio.h>:** The conio.h header file used in C programming language contains functions for console input/output. Some of its most commonly used functions are clrscr, getch, getche, kbhit etc. They can be used to clear screen, change color of text and background, move text, check whether a key is pressed or not and to perform other tasks.

2.1.1.1 Concepts of Pre-Compiler Directives

- Preprocessor is part of C compiler, which evaluates preprocessor directives in the final token stream passed to the compiler.
- It is a program that processes our source program before it is passed to the compiler.
- The preprocessor offers several features called preprocessor directives.
- Each of these preprocessor directives begins with a # symbol. Only white space characters may appear before a preprocessor directive on a line.
- The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition.
- Preprocessor directives are lines included in the code of programs preceded by a hash sign (#).
- The following are the preprocessor Directives available in C:
  - Macro Definitions (#define, #undef)
  - File Inclusions (#include)
  - Conditional Compilation (#ifdef, #ifndef, #if, #endif, #else, #elif)

2.1.1.2 Use of #include and #define

#include

- The #include preprocessor directive is used to include (paste) code of given header file into current program file.
- It is used to include system-defined (built-in) and user-defined header files.
- If included file is not found, compiler gives an error.
- We can use #include in two ways:

Syntax : #include <file>

- This is used for system or built-in header files. It searches for a file named file in a list of directories specified by us, then in a standard list of system directories.

  E.g. : **#include<stdio.h>**

Syntax: #include "file"

- This is used for header files of our own program. It searches for a file named file first in the current directory, then in the same directories used for system header files.
- E.g. : **#include “MyHeader.h”**

#define:

- To define preprocessor macros we can use #define.
- **Its syntax is: #define identifier value**
- When the preprocessor finds this directive, it replaces any occurrence of identifier in the rest of the code by replacement.
- This replacement can be an expression, a statement, a block or simply anything.
- The preprocessor simply replaces any occurrence of identifier by replacement.
- Before we can use a macro, we must define it explicitly with the `#define' directive. `#define' is followed by the name of the macro and then the code it represents.

  #define identifier replacement-text
- When this line appears in a file, all occurrences of identifier will be replaced by the replacement-text automatically before the program is compiled.
- E.g. **#define PI 3.14**

  The above statement defines a macro named “**PI**” as a short form of 3.14. If there is a statement in our C program as follows: **area = PI * r * r;** then the C Preprocessor will identify and replace the macro name **PI** with value 3.14.

Benefits of using #define

- Use of Uppercase names is a standard convention. Programs are easier to read.
- The macro name PI is known as a “Symbolic Constant”
- Symbolic constants enable the programmer to create a name for a constant and use the name throughout the program.

- The advantage is, it is only needed to be modified once in the #define directive, and when the program is recompiled, all occurrences of the constant in the program will be modified automatically, making writing the source code easier in big programs.

Macro with Arguments

- We can also define macros with arguments just like functions.
- #define identifier(f1,f2,f3…fn) replacement-text
- Here f1, f2 …fn are the formal arguments which will be replaced by the replacement-text (actual arguments) when the macro will be called.
- #define CUBE(X) (X * X * X)
- If the following statement, volume = CUBE(side); is found then it is replaced by the statement above and will be expanded as volume = (x * x * x);

Macro versus Functions

- A macro can be used to calculate the area of the circle. We know that, even a function can be written to calculate the area of the circle.
- Though macro calls are ‘like’ function calls, they are not really the same things.
- Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.
- The fact that macros use text replacement creates the potential for bugs.

```
#define DOUBLE(X) X*X

 int y = 3;
 int j = DOUBLE(++y);
```

- If we’re expecting that j will be assigned a value of 4 squared (16), then you would be wrong. Because of the text replacement, what actually happens is that the DOUBLE (++y) expands to ++y * ++y, which equals 4*5, giving us 20.

Macro	Function
Macro is **Preprocessed**	Function is **Compiled**
No Type Checking	Type Checking is Done
Code Length Increases	Code Length remains Same
Use of macro can lead to **side effect**	No **side Effect**
Speed of Execution is **Faster**	Speed of Execution is **Slower**
Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
Useful where small code appears many time	Useful where large code appears many time
Generally Macros do not extend beyond one line	Function can be of any number of lines
Macro does not Check **Compile Errors**	Function Checks **Compile Errors**

2.2 Input Output Statements

- I / O functions are grouped into two categories.
    - **Unformatted I/O functions**
    - **Formatted I/O functions.**

Unformatted I/O Functions

1. **getche( ): -** This function is used to read the character from the standard input device. The format of it is: **Syntax: variablename = getche( ) ;**
   e.g
   char name;
   name = getche( ) ;

- Here computer waits until we enter one character from the input device and assign it to character variable name.
- In getche( ) function there is no need to press enter key after inputting one character but we are getting next result immediately while in getchar( ) function we must press the enter key.
- This getche( ) function is written in standard library file 'conio.h'.

2. **getchar()** - **For reading a Single character (Unformatted I/P)**

- It is the simplest function to read a character from the 'standard input device (Keyboard)'.
  Syntax : var_name = getchar(); where var_name is a valid char type variable in C.
- When the above statement is executed in C, the computer waits until a key is pressed and then assigns the character as a value to getchar() which then assigns it to the variable on the left side.
  e.g
  char name;
  name = getchar( ) ;

3. **putchar() - For writing a Single character (Unformatted O/P)**

- It is used to write a character on standard output/screen.
  Syntax : putchar(var_name); where var_name is a valid char type variable in C.

Ex: Program to read a line of text from the user and display it on screen.

```
#include <stdio.h>
#include <conio.h>
void main()
{
  char ch;
  printf("Enter your character ");
  ch = getchar();
while (ch != '\n')
{
        putchar(ch);
        ch = getchar();
}
getch();
}
```

4. gets() – For reading entire line of text

- gets() is used to read the string (sequence of characters) from the input device (Keyboard).

Syntax: char *str; or char str[10];
gets(str); where str is an array of char, i. e., a character string.

- The function reads characters entered from the keyboard until newline is entered and stores them in the argument string str, The newline character is read and converted to a null character (\O) before it is stored in str.

5. puts() – For writing a line of text

- puts() is used to print a string on the display i.e. the screen.

Syntax: puts(str); where str is an array of char, i. e., a character string.

Ex: Program to read a line of text from the user and display it on screen.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        char str[81];
        puts("Enter a line of text:\n");
        gets (str);
        puts("You entered:\n")
        puts(str);
        getch();
}
```

- The above line of text can contain 80 characters. Thus, the array str has been declared to store 81 characters with a provision to store the null terminator ('\0').

6. getc() (Reading a character from a file)

- getc() reads a single character from a file.

Syntax: char-var = getc(file-ptr);

7. putc() (writing a character to a file)

- putc() writes a single character to a file.

Syntax: putc(character-var, file-ptr);

Formatted Input and Output Functions

- The Formatted I/O functions allow programmers to specify the type of data and the way in which it should be read in or written out.
- They are defined in the "stdio.h" header files.
- We have scanf() for input and printf() for output in C with different formatting options.

1. scanf() - Formatted Input

- Formatted input refers to an input data that is arranged in a particular format.

- **Scanf()** is used to take input from the standard input device in a formatted manner.
- We can read a character, string, numeric data from keyboard using this function.
  Syntax: scanf ("*control string*", arg1, arg2, arg3,….argn) ;

Here the control string specifies the field format in which the data is to be entered and the arguments arg1, arg2, ….argn specify the addresses of the locations where the data is stored.

- In the scanf, the ampersand (&) before the variables is a must.

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

Reading Integers

- scanf("%[w]d",arg); - Here 'w' is a optional number that specifies the width of the input and d specifies that the number will be an integer.

Eg.

int a, b;
scanf("%2d %5d", a, b); //If i/p is 12 and 54321 then it is assigned to a and b
but if scanf("%2d %5d", a, b); //If i/p is 54321 and 12 then 54 is assigned to a and 321 to b.
The number 12 will remain unread and it will be input to the next scanf() call.

- We can skip an input field by putting * in front of %d.
  scanf("%d %*d %d",&a, &b); - If we input 123 456 789 then 123 is assigned to a and 789 to b. 456 is skipped.
- For reading ***long integers*** we specify %ld and for short integers %hd.

Reading Floating point numbers (Real Numbers)

- To read a real-number we use specification %f.
  float a, b, c;
  scanf("%f %f %f", &a, &b, &c);
- For reading ***double data,*** we use %lf

Reading a single character

- To read a single character we use %wc. scanf("%[w]c", &ch); w – specifies width

Reading a string (Sequence of characters)

- To read a string we use %ws scanf("%[w]s", str); w – specifies width
  Here str is a pointer to a char variable or an array.
  Eg. char name[20];
  scanf("%s", name); I/p Good morning O/p Good
  The above statement reads characters. %s terminates reading when it finds a blank space.

Note:

- **scanf()** cannot read a string containing Spaces.

We can allow or reject particular characters from a string:

- To allow only selected characters:
  scanf("%[a-z]", name); - reads name with only alphabets
- Reject characters
  scanf("%^[\n]",name); - reads name till enter key is not pressed. Allows spaces also.

2. printf() - Formatted output

- **printf()** statement provides formatted output that can control the alignment and spacing of the output sent to the terminal.
  We can display an integer value using following syntax:
  printf("control string", arg1, arg2, arg3,….argn) ;
  - *control string specifies how many arguments follow and what are their types?. The arguments arg1, arg2, …..argn are the variables whose values are formatted and printed according to the field specified in the control string.*
  - *The arguments should match in number, order and type with the field specifier.*

<u>Output of Integers</u>

- We can display integers using the following:
  Syntax: printf("%[w]d",arg); - w specifies the minimum width of output

E.g. int x = 12, y = 3412;
printf("%4d",x);

		1	2

printf("%4d",y);

3	4	1	2

- If we want to display the value left justified, we put a minus sign before [w].

E.g. int x = 12;
printf("%-4d",x);

1	2		

- It is also possible to fill the leading spaces with zeroes by specifying 0 before w.

E.g. int x = 12;
printf("%04d",x);

0	0	1	2

<u>Output of Real Numbers</u>

- We can display real numbers using the following:
  Syntax: printf("%[w.p]f",arg); - w specifies the minimum width of output
  p is the number of digits after the precision
- If the number of digits is more than p, the value is rounded off up to 3 digits and display right justified in w columns.

E.g. float x = 98.7654;
printf("%f", x);

9	8	.	7	6	5	4			

printf("%10.2f", x);

					9	8	.	7	7

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

9	8	.	7	7					

printf("%10.2e", x);

		9	.	8	8	E	+	0	1

<u>Output of Characters</u>

- We can display characters using the following:
  Syntax: printf("%[w]c",arg); - w specifies the minimum width of output

E.g. char ch = 'A';
printf("%c", ch);

'A'

printf("%3c", ch);

		'A'

printf("%-3c", ch);

'A'		

Output of Strings

- We can display strings using the following:

Syntax: printf(“%[w.p]s”,arg); - w specifies the minimum width of output
- p specifies the first p of characters to be displayed in w.

E.g. char str[] = “Hello”;

printf(“%s”, str);

‘H’	‘E’	‘L’	‘L’	‘O’

printf(“%10s”, str);

					‘H’	‘E’	‘L’	‘L’	‘O’

printf(“%-10s”, str);

‘H’	‘E’	‘L’	‘L’	‘O’					

printf(“%10.2s”, str);

								‘H’	‘E’

Note:

- We can also specify user-defined width [w.p] in printf().

```
printf(“%*.*s”,w,p,str);
void main()
{
        int i;
        char str[10];
        printf(“\nEnter a String:-“);
        scanf(“%s”, str);

        for(i=0; str[i] !=’\0’ ; i++)
        {
                printf(“%-*.*s\n”, 10, i+1, str);
        }
        getch();
}
```

The following are the specifier which can be used in C with both printf() and scanf().

Field -Specifier	Description
%c	Single character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	Signed integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)

%ld	Long integer
%hd	Short integer
%lf	Double
%Lf	Long double

2.3 Operators:

- An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.
- C language is rich in built-in operators and provides the following types of operators −
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Increment/ Decrement Operators
  - Special Operators

Arithmetic Operators

- The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

Relational Operators

- The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

- Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then −

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

- Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

P	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

- Assume A = 60 and B = 13 in binary format, they will be as follows –

A =	0	0	1	1	1	1	0	0		A =	0	0	1	1	1	1	0	0
B =	0	0	0	0	1	1	0	1		B =	0	0	0	0	1	1	0	1
A & B =	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**		**A | B =**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**
A =	0	0	1	1	1	1	0	0		A =	0	0	1	1	1	1	0	0
B =	0	0	0	0	1	1	0	1		**~A =**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**
A ^ B =	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**										

- The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101

<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

- The following table lists the assignment operators supported by the C language −

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Special Operators

Operator	Description	Example

sizeof()	Returns the size of a variable/Datatype.	**int a;**
		x = sizeof(a) will have x = 2. As **a** takes 2 bytes

&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression/ Ternary Operator	If Condition is true ? then value X : otherwise value Y

		E.g. X = (a > b) ? a : b; **Here if a>b is true then X = a else X = b**

Increment/ Decrement Operators

- C has two special unary operators called increment (++) and decrement (--) operators. These operators increment and decrement value of a variable by 1.

```
++x is same as x = x + 1 or x += 1
--x is same as x = x - 1 or x -= 1
```

- Increment/Decrement operators are of two types:
  - **Prefix increment/decrement operator.**
  - **Postfix increment/decrement operator.**

Prefix increment/decrement operator.

- The prefix increment/decrement operator immediately increases or decreases the current value of the variable. This value is then used in the expression. E.g. If x = 5, then

```
y = ++x;        // will give the answer y = 6 and
y = --x;        // will give the answer y = 4
```

- Here first, the current value of x is incremented or decremented by 1. The new value of x is then assigned to y.

Postfix increment/decrement operator.

- The postfix increment/decrement operator uses the current value in the expression and then increases or decreases the value of the variable. E.g. If x = 5, then

```
y = x++;        // will give the answer y = 5 and
y = x--;        // will give the answer y = 5
```

- Here first, the current value of x is assigned to y and then value of x is incremented or decremented by 1.

Arithmetic Expressions and Evaluation of Arithmetic Expressions (Operators Precedence in C)

- An expression is a sequence of operands and operators that reduces to a single value.
- If **a** and **b** are integers then a + b is called **integer arithmetic expression** and the result is an integer.
- If **a** and **b** are real numbers then a + b is **Real arithmetic expression** and the result is a float value.
- If **a is int** and **b is float** then a + b is **Mixed-mode arithmetic expression** and the result is a float value.
- We use the **operator precedence and associativity rules** to determine the value of an expression.
- **Operator precedence** determines the grouping of terms in an expression and decides how an expression is evaluated.
- **Precedence** represents the priority of operators.
- Arithmetic expression without parentheses ( ) will be evaluated from left to right using the rules of precedence of C operators.
- There are two levels of priority of arithmetic operators in C:
  - High Priority → * / %
  - Low Priority → + -
- The operators in an expression are evaluated in the order of their precedence.
- If the expression contains more than one operator at the same precedence level, they are associated with their operands using the associativity rules.
- Consider the Expression : **X = a - b / 3 + c *2 - 1.**

  Let a = 9, b = 12, and c = 3
- The steps to convert the given valid C expression to its mathematical form (if possible) and to evaluate it are as follows:

Step 1: x = 9 – 12 / 3 + 3 * 2 – 1

Step 2: x = 9 – 4 + 3 * 2 – 1

(Here / has higher priority + and - so left to right firstly division will be solved)

Step 3: x = 9 – 4 + 6 – 1

(Here * has higher priority than + and - so left to right firstly multiplication will be solved)

Step 4: x = 5 + 6 – 1

Step 5: x = 11 – 1

Step 6: x = 10

- Again consider the Expression : **X = a - b / (3 + c) *(2 - 1).**

  Let a = 9, b = 12, and c = 3
- When parentheses are used, the expressions within the parentheses have the highest priority.

- If two or more parentheses ( ) are used in expression, then the expression in the left-most parentheses and then the right-most in the last.
- If parentheses are nested (parentheses within parentheses) then the innermost parentheses will be solved first.
- The steps to convert the given valid C expression to its mathematical form (if possible) and to evaluate it are as follows:

Step 1: x = 9 – 12 / (3 + 3) * (2 – 1)

Step 2: x = 9 – 12 / 6 * (2 – 1)

(Here 1st ( ) has higher priority so left to right firstly () will be solved)

Step 3: x = 9 – 12 / 6 * 1

(Here 2nd ( ) has higher priority so left to right () will be solved)

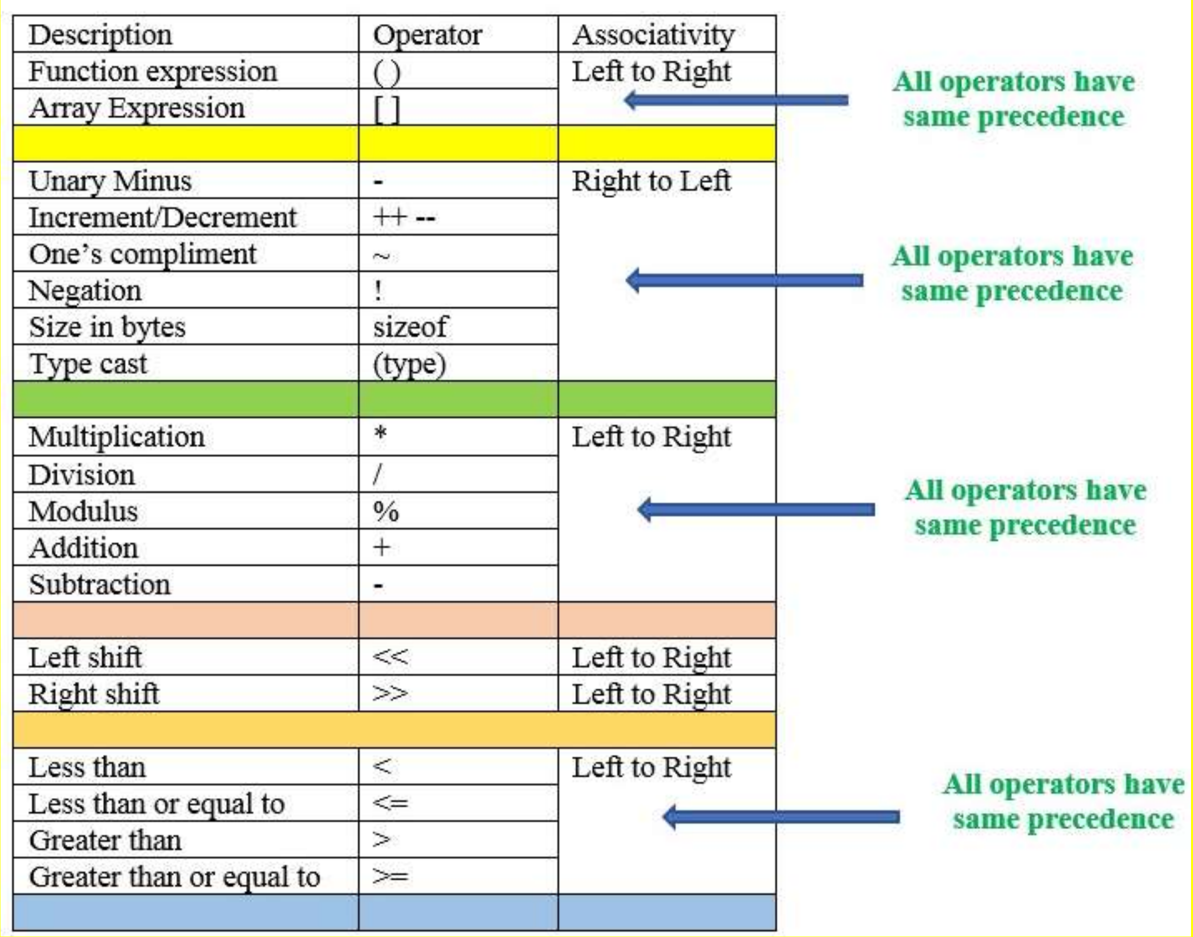
Step 4: x = 9 – 2 * 1

(Here **/** has the highest priority so from Left to right division is performed.)

Step 5: x = 9 – 2

(Here * has the highest priority so from Left to right Multiplication is performed.)

Step 6: x = 7



Description	Operator	Associativity
Function expression	( )	Left to Right
Array Expression	[ ]	
Unary Minus	-	Right to Left
Increment/Decrement	++ --	
One's compliment	~	
Negation	!	
Size in bytes	sizeof	
Type cast	(type)	
Multiplication	*	Left to Right
Division	/	
Modulus	%	
Addition	+	
Subtraction	-	
Left shift	<<	Left to Right
Right shift	>>	Left to Right
Less than	<	Left to Right
Less than or equal to	<=	
Greater than	>	
Greater than or equal to	>=	

All operators have same precedence

All operators have same precedence

All operators have same precedence

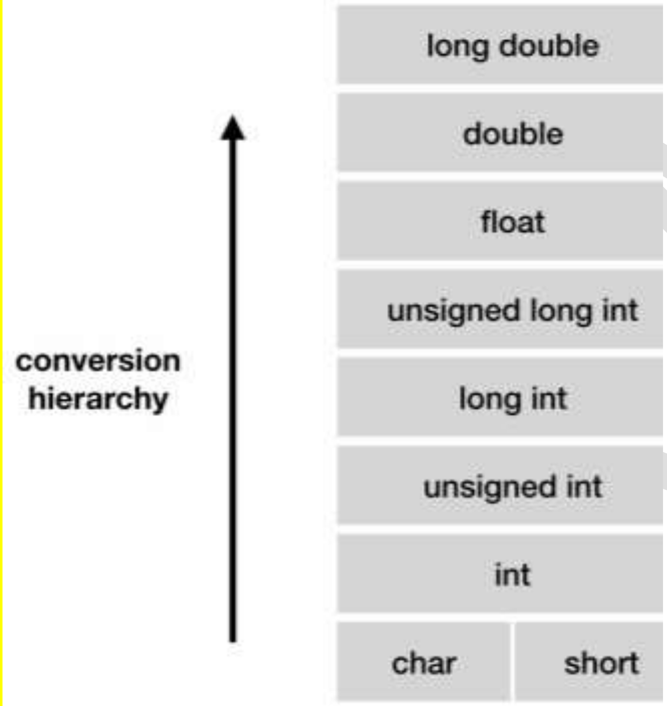
All operators have same precedence

Type Conversion in C

- Typecasting is converting one data type into another one.
- It is also called as data conversion or type conversion.
- It is one of the important concepts introduced in 'C' programming.
- 'C' programming provides two types of type casting operations:

Implicit Type Conversion

- Implicit type casting means conversion of data types without losing its original meaning.
- It happens automatically when a value is copied to its compatible data type.
- During conversion, strict rules for type conversion are applied. If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher data type.
- E.g. int a; float b; then a + b will be automatically converted to float type



- **We cannot perform implicit type casting on the data types which are not compatible with each other such as:**
- Converting float to an int will truncate the fraction part hence losing the meaning of the value.
- Converting double to float will round up the digits.
- Converting long int to int will cause dropping of excess high order bits.

Explicit Type Conversion

- There are some situations in which we may have to force type conversion.
- Suppose we have a variable div that stores the division of two operands which are declared as an int data type. E.g. int a, b; float c; then c = a/b will give int because though c is int but **(a/b)** is an int expression so to convert the result explicitly to float we use type casting.

  Syntax: (type-name) Expression

  E.g. c = (float) a / b; // will give float ans

2.4 Built-in Functions:

2.4.1 Use of <string.h>:

- ‘C’ library has many functions for handling strings. These functions are available in **string.h** header file.
- The following are the most commonly used String functions:
    1. strlen()
    2. strcpy(), strncpy()
    3. strcmp(), strncmp()
    4. strcat(), strncat()
    5. strrev()

1. **strlen():** This function counts and returns the number of characters in a string.

   Syntax: l = strlen(str); // l is an integer variable and str is a character variable.

   The counting of characters end when it finds the first null character.

   E.g.

   ```
   int l;
   char str[]=”Hello”;
   l = strlen(str);
   printf("%d", l);          // o/p is 5
   ```

2. **strcpy() and strncpy() :**

- We cannot copy a string variable directly to another variable using ‘=’ operator. Strcpy() copies the one string to another.

  Syntax: strcpy(str1, str2); // Where str2 is the source string and str1 is the destination string. It copies value of str2 to str1.

strncpy() – This function copies the leftmost ‘n’ characters of str2 into str1.

Syntax: strncpy(str1, str2, n);

// str1 and str2 are two strings and n is the no. of characters to be copied from the left side

E.g.

```
char str1[10] = “Hello”;
char str2[10];
char str3[10];
strcpy(str2, str1);                   // copies str1 to str2
strncpy(str3, str2, 2);
printf(“\n str2 = %s”, str2);               // O/p is Hello
printf(“\n str3 = %s”, str3);              // O/p is He
```

Note: size of the string1 (Destination string) should be large enough to hold the string.

3. **strcmp() – String Comparison and strncmp()**

- The strcmp() function compares two strings and returns 0 if both strings are equal.
- The strcmp() function takes two strings and return an integer.
- The strcmp() compares two strings character by character. If the first character of two strings is equal, next character of two strings is compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

- If the characters are not equal, it returns the difference of ASCII value of the first non-matching characters.

Syntax: **var =** strcmp(str1, str2); // str1 and str2 are two strings (**var is int)**

strncmp() – This function compares the leftmost ‘n’ characters of str1 with str2.

Syntax: var = strncmp(str1, str2, n);

// str1 and str2 are two strings and n is the no. of characters to be compared from the left side

```
#include <stdio.h>
#include <string.h>
void main()
{
   char str1[] = "abcd", str2[] = "abCd";
   int result;
   // comparing strings str1 and str2
   result = strcmp(str1, str2);
   printf("Ans is %d\n", result);                 // O/p – 32      c - C

   // comparing strings str1 and str2
   result = strncmp(str1, str2, 2);
   printf("Ans is %d\n", result);                 // O/p 0 (First 2 chars of Strings r equal)

getch();
}
```

4. strcat() – String Concatenation and strncat()

- The process of joining two strings is called Concatenation.
- We cannot join strings using ‘+’ operator.
- To join two strings, first we copy str1 in str3 and then append str2 at the end of str3.

Syntax: strcat(str1, str2); // str1 and str2 are two strings

- When the function is about to execute, str2 is appended to str1 by removing the null character at the end of str1 and placing str2 from there. Str2 remains unchanged.

strncat(): It concatenates the first ‘n’ characters of str2 to str1.

Syntax: strcat(str1, str2, n);

// str1 and str2 are two strings and n is the no. of chars to be concatenated from the left side

```
#include <stdio.h>
#include <string.h>
void main()
{
     char str1[10] = "very", str2[] = "good”, str3[20];
     strcpy(str3, str1);
```

```
      strcat(str3, str2);
      printf("str3 is %s\n", str3);                    // O/p – very good
      strncat(str3, str2, 4);
      printf("\nstr3 is now %s", str3);               //o/p - very good good
getch();
}
```

5. **strrev()**

- **strrev()** function is used to reverse a given string.

```
#include <stdio.h>
#include <string.h>
void main()
{
      char str1[10] = "Morning", ch[10];
      ch = strrev(str1);
      printf("\nReverese of  %s is now %s", str1, ch);
      //o/p -     Reverse of Morning is now gninroM
getch();
}
```

2.4.2 Use of <math.h>:

- The **math.h** header file defines various mathematical functions and one macro. All the functions available in this library take double as an argument and return double as the result.

1. **abs():** returns the absolute value of an integer. (Given number is an integer)
   Syntax: abs(number) E.g.: a = abs(-5); O/p: 5

2. **fabs():** returns the absolute value of a floating-point number. (Given number is a float number)
   Syntax: fabs(number) E.g.: a = fabs(-5.3);O/p: 5.3

3. **pow():** Raises a number (x) to a given power (y).
   Syntax: double power(double x, double y) E.g. : y = pow(3, 2); O/p: 9

4. **sqrt() :** Returns the square root of a given number.
   Syntax: double sqrt (double x) E.g. : y = sqrt(9) O/p: 3

5. **ceil() :** Returns the smallest integer value greater than or equal to a given number.
   Syntax: double ceil(double x) E.g. y = ceil(123.54) O/p: y = 124

6. **floor() :** Returns the largest integer value less than or equal to a given number.
   Syntax: double floor(double x) E.g. y = floor(123.54) O/p: y = 123

7. **round():** round() function in c is used to return the nearest integer value of the float/double/long double argument passed to this function. If the decimal value is from ".1 to .5", it returns an integer value which is less than the argument we passed and if the decimal value is from ".6 to .9", the function returns an integer value greater than the argument we

passed. In short the round() function returns an integer value nearest to the argument we passed.

Syntax: double round(double x) **E.g. y = round(4.5)** **O/p: 4**
Syntax: double round(double x) **E.g. y = round(4.7)** **O/p: 5**

8. trunc(): We can use trunc() function whenever we need to calculate the integer part from a double data type. The advantage of this function is that whatever the decimal value may be the integer part remains same. In the ceil or floor or round functions the integer value changes but in trunc function it does not.
Syntax: double trunc(double x) **E.g. y = trunc(3.9)** **O/p: 3**

9. log(): Returns the natural logarithm of a given number.
Syntax: double log(double x) **E.g. y = log(1.00)** **O/p: 0.0000**

10. exp(): Returns the value of e raised to the x^{th} power.
Syntax: double exp(double x) **E.g. y = exp(1.00)** **O/p: 2.718282**

11. cos(): Returns the cosine of a radian angle x.
Syntax: double cos(double x) **E.g. y = cos(60)** **O/p: 0.500000**

12. sin(): Returns the sine of a radian angle x.
Syntax: double sin(double x) **E.g. y = sin(45)** **O/p: 0.707107**

13. tan(): Returns the tangent of a radian angle x.
Syntax: double tan(double x) **E.g. y = tan(45)** **O/p: 1.619775**

Functions in CTYPE.H

The ctype header is used for testing and converting characters. A control character refers to a character that is not part of the normal printing set.

The **is...** functions test the given character and return a nonzero (true) result if it satisfies the following conditions. If not, then 0 (false) is returned.

1. **isalnum() –** Checks whether a given character is a letter (A-Z, a-z) or a digit(0-9) and returns true else false. **Syntax: int isalnum(int character);**

2. **isalpha() –** Checks whether a given character is a letter (A-Z, a-z) and returns true else false. **Syntax: int isalnum(int character);**

3. **isdigit() –** Checks whether a given character is a digit (0-9) and returns true else false. **Syntax: int isdigit(int character);**

4. **iscntrl() –** Checks whether a given character is a control character (TAB, DEL)and returns true else false. **Syntax: int iscntrl(int character);**

5. **isupper() –** Checks whether a given character is a upper-case letter(A-Z) and returns true else false. **Syntax: int isupper(int character);**

6. **islower() –** Checks whether a given character is a lower-case letter(a-z) and returns true else false. **Syntax: int islower(int character);**

7. **isspace() –** Checks whether a given character is a whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed)and returns true else false.
   Syntax: int isspace(int character);

8. **tolower() –** If the character is an uppercase character (A to Z), then it is converted to lowercase (a to z).
   Syntax: int tolower(int character); E.g.: tolower('A') O/p : a

9. **toupper() –** If the character is a lowercase character (a to z), then it is converted to uppercase (A to Z).
   Syntax: int toupper(int character); E.g.: toupper('a') O/p : A

10. **toascii() –** Converts a character into a ascii value.
    Syntax: short toascii(short character);

Unit 3: Decision-Making Statements

- C Program is a set of statement executed sequentially under normal conditions.
- Sometimes we need to change the order of execution based on some condition. This includes decision making to check whether the condition statement is true or not and based on that execute the statements following the condition.
- These condition statements are known as Decision making statements. Since they also change or control the flow of execution, they are also called “Control Structures” or “Branching Structures”.
- They are called branching because the program follows one branch or another.
- C has the following Decision-making statements:
  - **if** statement
  - **switch** Statement
  - Conditional Operator (? :)
  - **goto** Statement

3.1. IF statements

- The “if” statement is a powerful decision making statement and is used to control the flow of execution of statements following it.
- The if statement can be written in 4 ways:
  - Simple if
  - if…else
  - Nested if
  - Else if ladder

3.1.1. Simple if

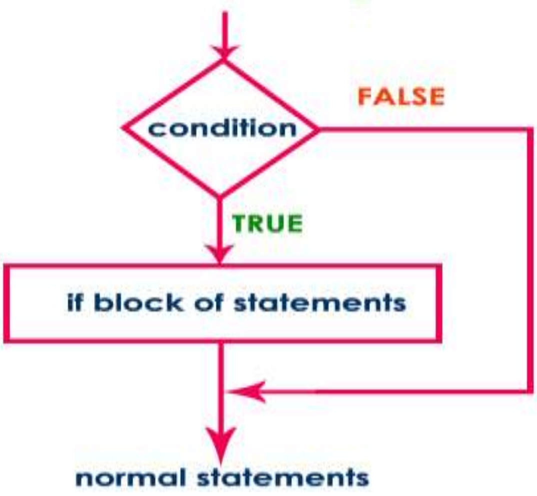
- The general form of simple if statement is

```
Syntax:     if(test expression)
            {
                    Statement-block;
             }
            statement-x;
```

- The statement-block may be a single statement or a group of statements.
- If the test expression is true the statement block will be executed and then statement-x will be executed. If it is false the statement-block will be skipped and the execution will jump to the statement –X.

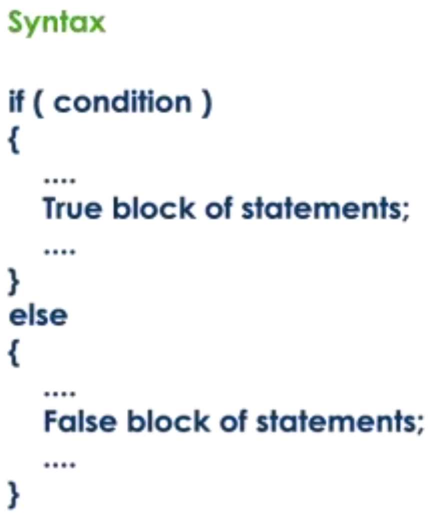
Example: Division will be performed only if b is non-zero.

```
    void main()
    {
        int a,b;
        float c;
        printf(“\nEnter A:-“);
        scanf(“%d”, &a);
        printf(“\nEnter B:-“);
        scanf(“%d”, &b);
        if ( b != 0 )
        {
                    c = (float) a / b;
        }
        printf(“\nAns is %d”, c);   }
```



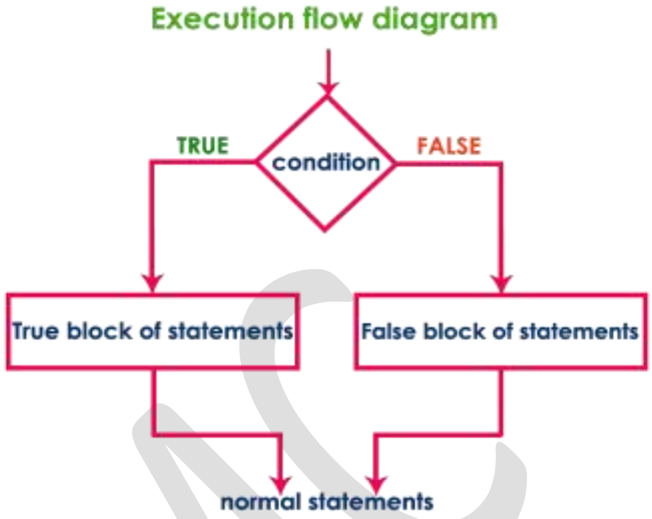
3.1.2. if…else

- The if..else statement is an extension of the simple If statement. The general form is:



Syntax

```
if ( condition )
{
   ....
   True block of statements;
   ....
}
else
{
   ....
   False block of statements;
   ....
}
```



- If the test expression is true then true-block statements are executed otherwise the false-block statements are executed.
- In both cases either true-block or false-block will be executed but not both.
- In both the cases, the control will then be transferred to statement-x.

Example: Program to check whether a number is odd or Even.

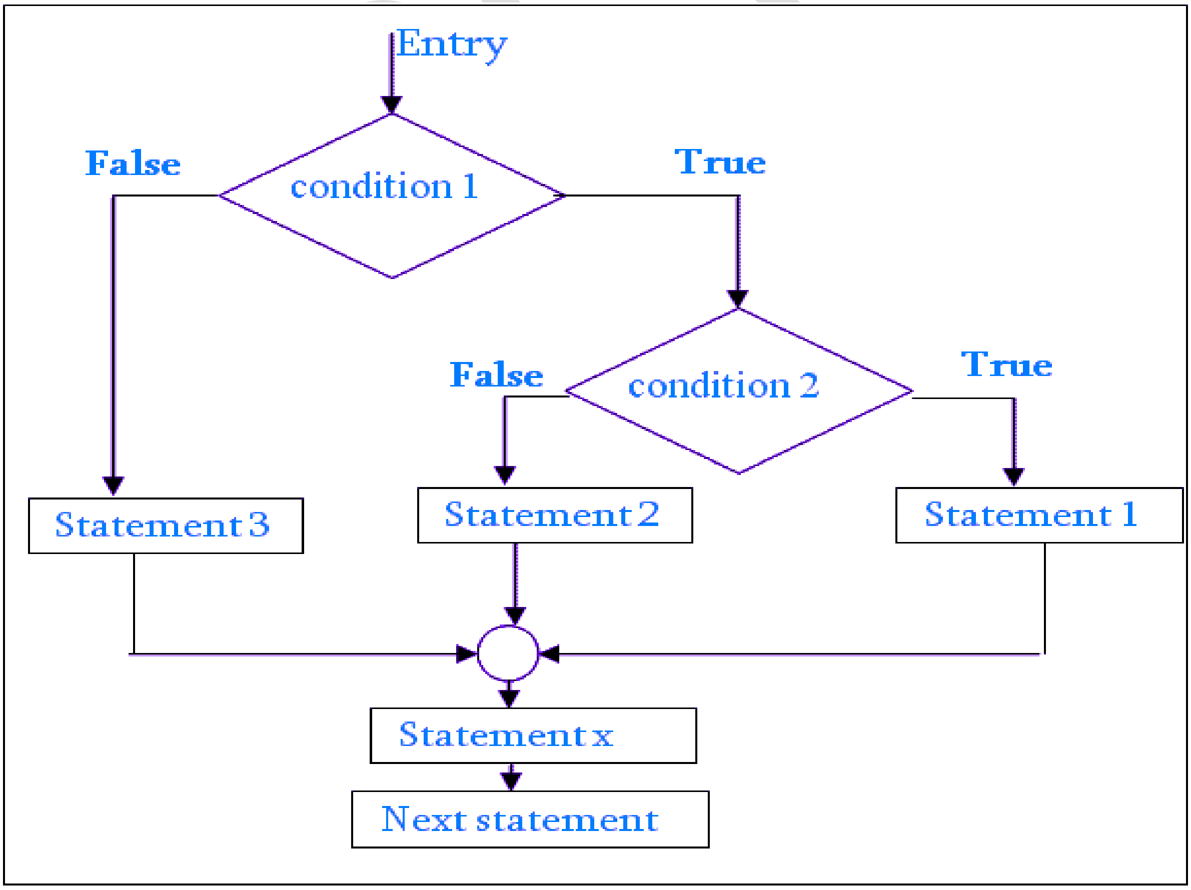
```
void main()
{
      int n;
      printf("\nEnter N:-");
      scanf("%d", &n);
      if ( n % 2 == 0 )
      {
              printf("%d is even", n);
      }
      else
      {
              printf("%d is odd", n);
      }
      getch();
}
```

3.1.3. Nested If

- When a series of decisions are involved we may have to use **more than one** if-else statement in nested form.
- If the ***test expression-1*** if false the statement-block 3 will be executed otherwise it continues to perform the nested if –else structure (inner part).
- If the ***test expression-2*** is true the statement-block 1 will be executed otherwise the statement-block 2 will be executed and then the control is transferred to the statement-x.

Syntax

```
if ( condition1 )
{
    if ( condition2 )
    {
        ....
        True block of statements 1;
    }
    ....
}
else
{
     False block of condition1;
}
```



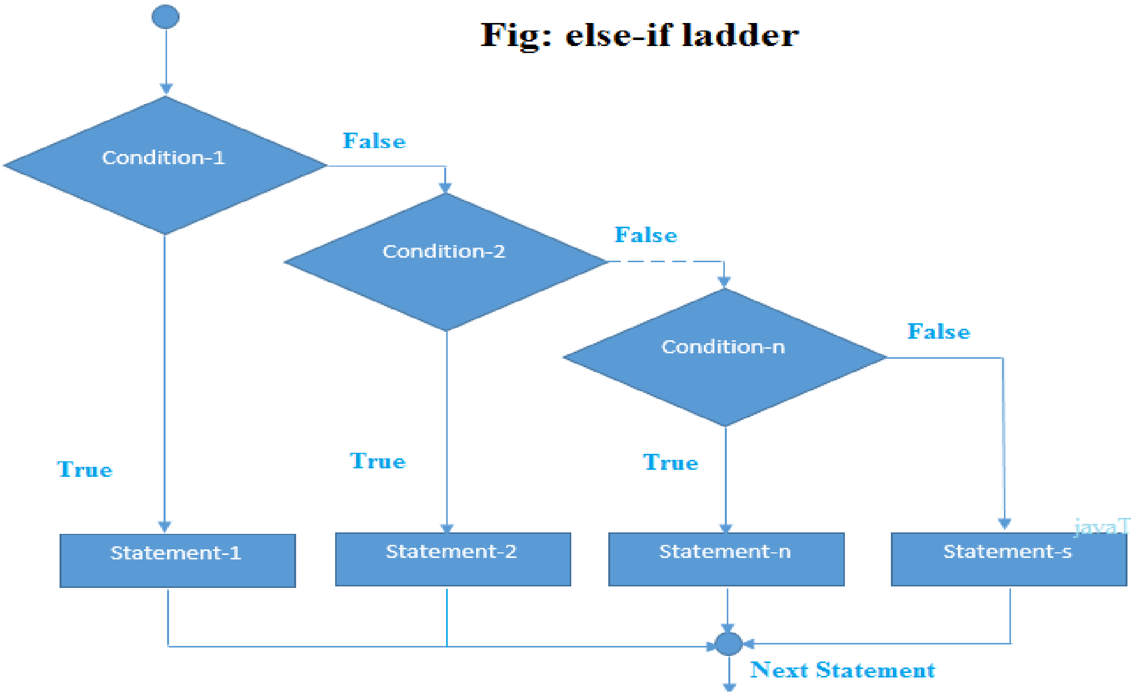
Example: Program to display greatest of 3 numbers

```
void main()
{
    int a,b,c;
    printf("\nEnter A:-");
    scanf("%d", &a);
    printf("\nEnter B:-");
    scanf("%d", &b);
    printf("\nEnter C:-");
    scanf("%d", &c);
    if ( a > b ){
        if( a > c )
        {
            printf("%d is >", a);
        }
        else
        {
            printf("%d is >", c);
        }
    }
    else
    {
        if( b > c )
        {
            printf("%d is >", b);
        }
        else
        {
            printf("%d is >", c);
        }
    }

getch();
}
```

3.1.4. Else if ladder

- When multiple conditions are to be checked, we have to use more than one if statements in a series. This forms a chain of if..else statements.
- This is known as if…else ladder because the conditions are executed from top to bottom in a step-wise manner.

Fig: else-if ladder



```
if(condition 1)
{
     statement-block1;
}
else if(condition 2)
{
     statement-block2;
}
else if(condition 3)
 {
     statement-block3;
 }
 …
 …
 …
 else if(condition n)
 {
     statement-blockn;
 }
 else
 {
     default-statement;
 }
 Statement-x;
```

- Here the conditions are evaluated from top to bottom. As a condition is true, the statements inside it are executed and control is transferred to statement-x skipping the remaining ladder.
- If condition-1 is true then statement-block 1 will be executed otherwise condition-2 is checked, if it is true statement-block 2 will be executed otherwise it move to condition-3 and then so on.
- If all the n conditions are false, the default-statement block will be executed. It is an optional else block. It may or may not be present every time.

Example: Program to check whether a number is positive, negative or zero

```
void main()
{
     int a;
     printf("Enter a Number: ");
     scanf("%d",&a);
     if(a > 0)
     {
             printf("Number is Positive");
     }
     else if(a == 0)
      {
          printf("Number is Zero");
     }
     else if(a < 0)
     {
          printf("Number is Negative");
     }
getch();
}
```

3.2. Switch case statement

- When we have multiple choices for selection, we use else…if ladder but when the number of choices increase, the complexity of program also increase.
- Thus, program becomes difficult to read and understand.
- ‘C’ has an alternative for else..if ladder known as switch.
- It is a mutliway decision statement. It tests the value of a given variable against a list of case values and when a match is found, a block of statements inside that case block are executed.

```
switch(expression)
{
    case value-1:
            block-1;
            break;
    case value-2:
            block-2;
            break;
    case value-3:
            block-3;
            break;
    case value-4:
            block-4;
            break;
    default:
            default-block;
            break;
}
Statement-x;
```

- The expression is an integer expression or character. *Value-1, Value-2*,… are constants or constant expressions (evaluated to an int)and are known as case-labels.
- The case-labels end with a colon (:)
- The **break** statement at the end of each block represents the end of the block and helps to exit from the switch statement and transfer to statement-x.
- If we do not use break statement at the end of each case, program will execute all remaining case statements until it finds next break statement or till the end of switch case block.
- The **default** block is optional. If all the case blocks do not match with the value of expression, the **default** block is executed. It can be placed anywhere between the switch statement but keeping it at the end is more preferable.

Note:

- We can also combine multiple case labels.
- We can also have nested Switch (switch inside switch).
- We cannot pass any value other than int or char constant as an expression.

Eg: Menu driven Program for Basic calculator using Switch statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b,c, choice;
   clrscr();
               printf("\n 1. Press 1 for Addition");
               printf("\n 2. Press 2 for Subtraction");
               printf("\n 3. Press 3 for Multiplication");
               printf("Enter 2 numbers:");
                scanf("%d%d",&a,&b);
               printf("\n Enter your choice:");
               scanf("%d",&choice);

               switch(choice)
               {
                     case 1,5:
                                   c=a + b;
                                    printf("Addition is %d", c);
                                    break;
                     case 2:
                                   c=a - b;
                                    printf("Subtraction is %d", c);
                                   break;

                    case 3:
                                   c=a * b;
                                    printf("Multiplication is %d", c);
                                    break;


               }

   getch();
}
```

3.2.1. Use of break and default

break:

- The **break** statement is used inside the switch to terminate a statement sequence.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- The break statement is optional. If omitted, execution will continue on into the next case.
- The flow of control will continue to cases until a break is reached.

default:

- A switch statement can have an optional default case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true.
- No break is needed in the default case.

3.2.2. Difference between if and Switch case

Else..if	switch
In *else if ladder*, the control goes through the every else if statement until it finds true value of the statement or it comes to the end of the else if ladder.	In case of switch case, as per the value of the switch, the control jumps to the corresponding case.
The if..else statement is not very compact and readable.	The switch statement is very compact and readable.
There is no need of **break** in else..if ladder.	There is need of **break** in every case block.
We can use expressions or variables of any type in else..if.	We can only use integer constants and character constants in switch.
It is more flexible because we can test a single expression for a set of values.	It is less flexible because for every value we need a single case block.
Each else..if block is dependent on each other and are executed in a fixed order.	Each **case** block is independent of other case blocks.
Else..if works on basis of true or false.	Switch works on the basis of equality (==).

Conditional Operator (? :)

- It is the only ternary operator available in C. It has 3 operands

  Syntax: var = (condition Exp) ? exp1 : exp2;
- The conditional operator works as follows:
  - conditional Expression is evaluated first. This expression evaluates to 1 if it's true and evaluates to 0 if it's false.
  - If conditionalExpression is true, exp1 is evaluated and value is assigned to var.
  - If conditionalExpression is false, exp2 is evaluated and value is assigned to var.

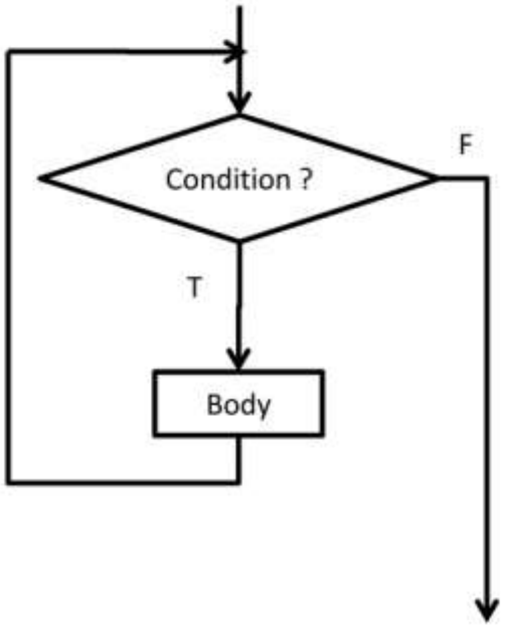
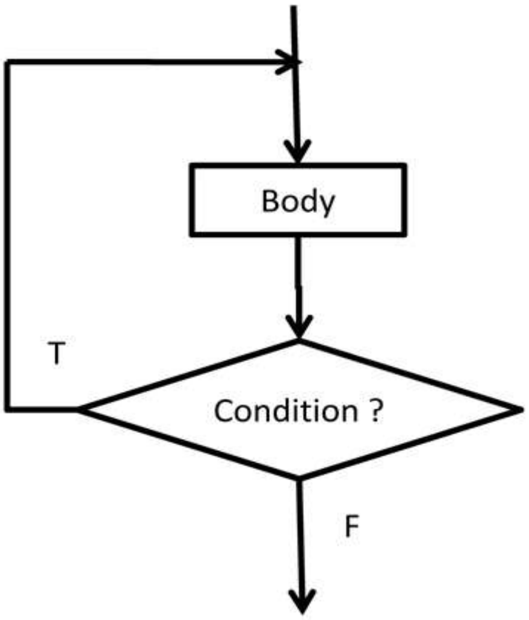
Eg: to Find max of two numbers

```
void main()
{
        int x, y, max;
        printf("\nEnter X:-");
        scanf("%d" , &x);
        printf("\nEnter Y:-");
        scanf("%d" , &y);
        max = (x < y) ? x : y;
        printf("\nMax is %d", max);
```

```
                getch();
        }
```

Unit 4: Iterative Statements

- Sometimes we want to execute a set of instructions repeatedly. This repetition of instructions is called looping.
- Looping can be done in such a way that the instructions are executed for fixed number of time based on some condition. If there is no condition than the loop will execute infinite number of times.
- In ‘C’, there are two types of loops depending on where the condition is for controlling the loop will be placed.
    - Entry-controlled loop
    - Exit-controlled loop

Entry Controlled Loop	**Exit Controlled Loop**
Test condition is checked first, and then loop body will be executed.	Loop body will be executed first, and then condition is checked.
If Test condition is false, loop body will not be executed.	If Test condition is false, loop body will be executed once.
for loop and while loop are the examples of Entry Controlled Loop.	do while loop is the example of Exit controlled loop.
Entry Controlled Loops are used when checking of test condition is mandatory before executing loop body.	Exit Controlled Loop is used when checking of test condition is mandatory after executing the loop body.
	

Note:

- Based on the nature of the variable and the value assigned to it, the loops can be classified into two categories:

- Sentinel-Controlled loops
- Counter-Controlled loops

Counter-Controlled Loop

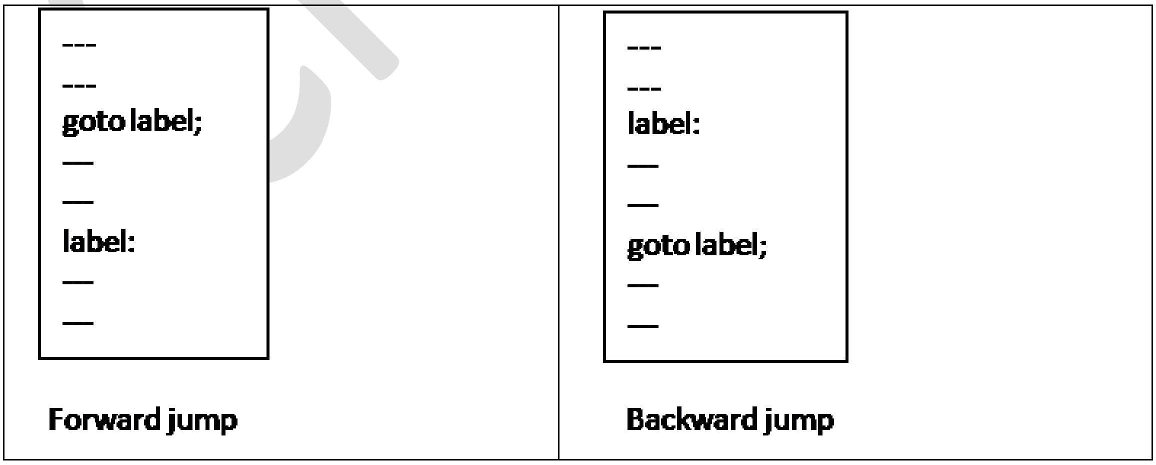
- When we know how many times loop body will be executed known as Counter Controlled Loop, for example - print natural numbers from 1 to 100, such kind of problem will be solved using counter controlled loop.
- Here we use a counter variable which is responsible for controlling how many times the loop will be executed. Such a variable is called a counter variable and it is initialized with a constant value or variable before the loop begins.

Sentinel-Controlled Loop

- When we don't know exactly know how many times loop body will be executed known as Sentinel Controlled Loop, for example - Reverse a given number, such kind of problem will be solved using sentinel controlled loop.
- Here, a special value is used to control the loop. This value is known as the sentinel and the variable is known as sentinel variable. Eg: -1, -999, etc.

4.1 Use of goto for iteration

- C Supports an unconditional branching statement known as **goto** which is used to jump from one statement to another with or without using any condition.
- **goto** statement is used for changing the normal sequence of program execution by transferring control to some other part of the program.
- It requires a **label:** in order to identify where to jump.
- The label can be any valid identifier with a ':'.
- It is placed immediately before the statement where we want to jump.
- When, the control of program reaches to **goto** statement, the control of the program will jump to the **label:** and executes the code below it.
- We have two types of jumps using **goto:**
    - Forward Jump
    - Backward Jump



```
---
---
goto label;
—
—
label:
—
—
```

Forward jump

```
---
---
label:
—
—
goto label;
—
—
```

Backward jump

Forward Jump	Backward Jump(Use of goto for iteration)
Here the **label:** is placed somewhere after the **goto** statement and so some of the statements between the **goto** and **label:** will be skipped. Because it moves forward skipping some statements it is called Forward Jump.	Here the **label:** is placed before the **goto** statement and so a loop will be formed. It will let some of the statements between the label and **goto** to be repeated indefinitely. Because it moves backward repeating some statements, it is called Backward jump.
In Forward Jump, we do not need a condition compulsorily.	In Backward Jump we need a condition between goto and label compulsory to break the loop. Hence, we use **break** statement to exit from the loop.

Forward Jump	Backward Jump

void main()
{
int n;
clrscr();
printf(“\nEnter N:-“);
scanf(“%d” , &n);
if(n > 100)
goto msg;
printf(“%d”, n);
goto end;
msg:
printf(“\nInvalid Number”);
end:
getch();
}
	void main()
	{
	int n, i = 1;
	clrscr();
	printf(“\nEnter N:-“);
	scanf(“%d” , &n);
	msg:
	printf(“%d”, i);
	i++;
	if(i <= n)
	goto msg;
	getch();
	}

Entry-Controlled Loops

- In ‘C’, the following are entry-controlled loops**:**
  - **While**
  - **For**

4.2 While

- It is a conditional control loop statement.
- The while is an entry controlled loop statement.
- In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.
- Because the condition is tested before the body, it is a pretest loop.

Syntax:	**Program to print 1 to N numbers using while**

var = intial-val;
while(condition)
{
……
……
Body of loop
……
……
[increment/decrement] var;
}
	void main()
	{
	int n, i = 1;
	clrscr();
	printf(“\nEnter N:-“);
	scanf(“%d” , &n);
	while(i <= n)
	{
	printf(“\n%d”, i);
	i++;
	}
	getch();
	}

- Here the variable is initialized with the value and it is used in the condition. If the condition is true, the loop execution begins and the statements inside the loop are executed.
- The variable is also incremented or decremented.
- The loop ends when the condition becomes false.
- We can also nested while loop i.e. while inside while.

4.3 For

- The ‘**for**’ loop is also an entry-controlled loop that provides a more short loop control structure.
- It executes a set of statements repeatedly until a particular condition is satisfied.

Syntax:

for(initialization ; test-condition ; increment/ decrement)
{
……
Body of loop
……
……
}

The execution of for is as follows:

- Initialization of control variables is done first using assignment statements (E.g. i = 1. Here ‘i’ is known as the loop control variable).
- Next the control variable value is tested in the test-condition. If the condition is true, the body of the loop is executed. If it is false, the loop ends and the next statement after ‘**for**’ get executed.
- After executing the loop body, the control will be transferred to increment/decrement section. Here the value of the control variable will be incremented or decremented. This process continues till the control variable satisfies the test condition.

Program: To display sum of 1 to N numbers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, i, s = 0;
    clrscr();
    printf(“\nEnter N:-“);
    scanf(“%d” , &n);
    for(i = 1;  i <= n;  i++)
    {
        s = s + i;
    }
    printf(“\nSum is - %d”, s);
getch();
}
```

Additional features of for loop

- More than one variable can be initialized in the initialization section. We can use multiple variables by separating it with commas.
- We can also have multiple increments/decrements. They are also separated by commas.

```
e.g.for( i= 1, x = n ;  i <= x;  i++,  x--)
    {                 printf(“\n%d - %d”, i , x);        }
```

- The test condition can have compound relational statement in condition section and it is not limited to control variable.

```
int sum = 0, i;
for( i= 0 ;  sum <= 100 && i <=20;  i++)
{
    sum = sum + i;
    printf(“\n%d - %d”, i, sum);
}
```

- We can also have expressions in assignment statements in initialization and increment/decrement sections.

```
for( i = n + 1 ;  i >= 1;  i = i - 2)
```

- All sections of a ‘**for’** loop are optional and can be omitted, if required. But semi-colons separating the sections are compulsory.

i = 1;
for(; i <= n; i++)
{
printf(“\n%d”, i);
}
	for(i = 1; ; i++)
	{
	if(i <= n)
	printf(“\n%d”, i);
	}
		for(i = 1; i <= n;)
		{
		printf(“\n%d”, i);
		i++;
		}

- We can also have a for loop without a body. There is a semi-colon immediately after for.

```
for (i = 1; i <= 100000; i++);
```

Such loops are used for time-delays. The semi-colon which doesn’t contain any statement is called a Null statement.

4.4 Do..while (Exit-Controlled Loop)

- ‘C’ has a single exit-controlled loop known as **‘do while’.**
- In this loop, the body of the loop gets executed before the condition is checked.
- As a result, even if the condition is false the loop will be executed atleast once.
- Because, the condition is checked at the last, it is known as an Exit-controlled loop or Post-test loop.

Syntax:	**Program to print sum of 1 to N no’s using do..while**

var = intial-val;
    do
    {
        ……
        ……
        Body of loop
        ……
        ……
        [increment/decrement] var;
    }while(condition);
	void main()
	{
	    int n, i = 1, s = 0;
	    clrscr();
	    printf(“\nEnter N:-“);
	    scanf(“%d” , &n);
	    do
	    {
	        s = s + i;
	        i++;
	    } while(i <= n);
	    printf(“\nSum is %d”, s);
	getch();
	}

4.5 Nested For, While and Do..while

- We can also have a loop inside another loop. When one loop has another loop in its body it is called nesting of loops.
- The nesting can be done up to any level.
- If we use nesting, it is necessary to indent the inner loop inside by giving tabs.
- We can have nesting in for, do..while and while loops.

Program: To display Multiplication Table (Nested For)	

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, row, col;
    clrscr();
    printf(“\nEnter N:-“);
    scanf(“%d” , &n);
	for(row = 1; row <= n; row++)
	    {
	        for(col = 1; col <= 10; col++)
	        {
	                printf(“%4d”, row * col);
	        }
	    printf(“\n”);
	    }
	getch();
	}

- In the above program, for loop is nested. For every value of row, the loop inside will run i.e. for row = 1, the col = 1 to 10 will be executed.

Program: To display Multiplication Table (Nested While)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, row, col;
    clrscr();
    printf(“\nEnter N:-“);
    scanf(“%d” , &n);
    row  = 1;
while( row <= n){
col = 1;
        while(col <= 10)
        {
                printf(“%4d”, row * col);
                col++;
        }
    row++;
    printf(“\n”);
}
getch();
}
```

Program: To display Multiplication Table (Nested Do..While)

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, row, col;
    clrscr();
    printf(“\nEnter N:-“);
    scanf(“%d” , &n);
    row  = 1;
do{
    col = 1;
    do{
                printf(“%4d”, row * col);
                col++;
        } while(col <= 10);
    row++;
    printf(“\n”);
} while( row <= n);
getch();
}
```

4.6 Jumping Statements break and continue

- Sometimes, while executing a loop, it is required to skip a part of the loop or to exit the loop as soon as certain condition becomes true. This is known as jumping out of loop.
- In ‘C’, this can be done using the following statements:
    - **goto**
    - **break**
    - **continue**

goto

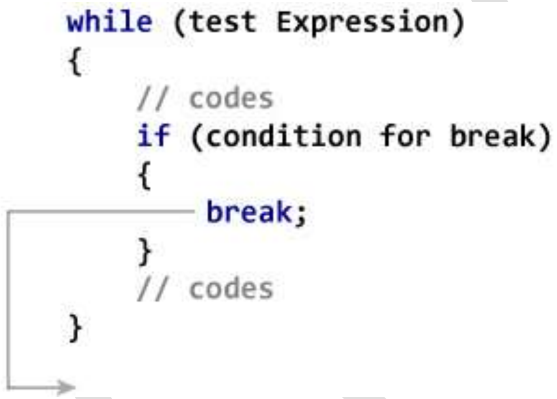
- We can make an exit from a loop using a **goto** statement in loop.
- Since, a goto statement can transfer the control to any place in the program unconditionally we can use it for branching.
- **goto** is very useful when we want to exit from deep nested loops.

Syntax:	**Program to print sum of 1 to N no’s using do..while**

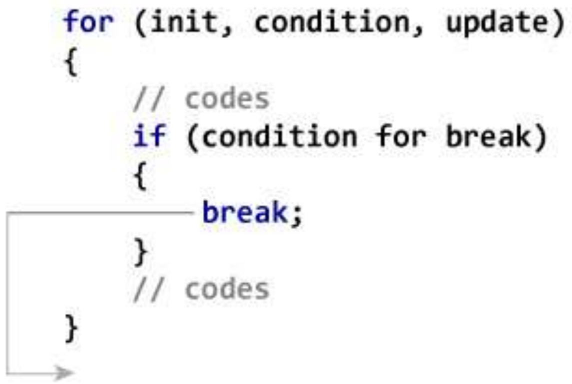
var = intial-val;
while(condition)
{
……
Body of loop
if(condition)
goto label;
……
……
[increment/decrement] var;
}
label:
	void main()
	{
	int n, i = 1, s = 0;
	printf(“\nEnter N:-“);
	scanf(“%d” , &n);
	while(i <= n) {
	if(i > 50)
	goto abc:
	s = s + i;
	i++;
	}
	abc:
	printf(“\nSum is %d”, s);
	getch();
	}

Break

- A **break** statement is used to exit from **for** loop, switch or if statement.
- Inside the loop when a **break** statement is found, it exits the loop immediately and continues from the next statement following the loop.
- When the loops are nested, the **break** statement will only exit the loop containing **break**. Therefore, break will exit only a single loop.



```
while (test Expression)
{
    // codes
    if (condition for break)
    {
      break;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
      break;
    }
    // codes
}
```

E.g.// Program to calculate the sum of maximum of 10 numbers

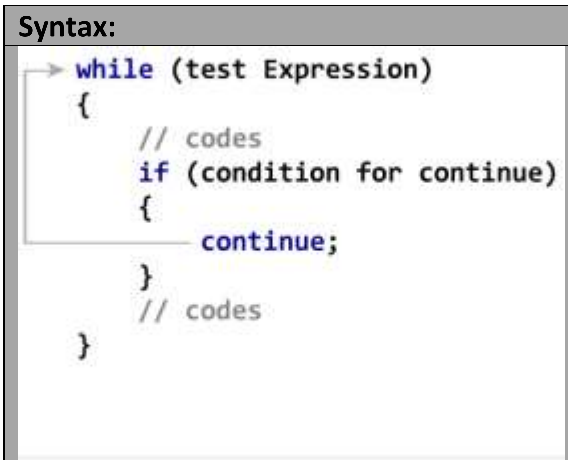
# include <stdio.h> 	
int main()
{
int i, number, sum = 0;
for(i=1; i <= 10; ++i)
{
printf("Enter n %d: ",i);
scanf("%d",&number);
	if(number < 0)
	{
	break;
	}
	sum += number; // sum = sum + number;
	}
	printf("Sum = %d",sum);
	getch();
	}

The above program will give the sum till the user enters a negative number.	

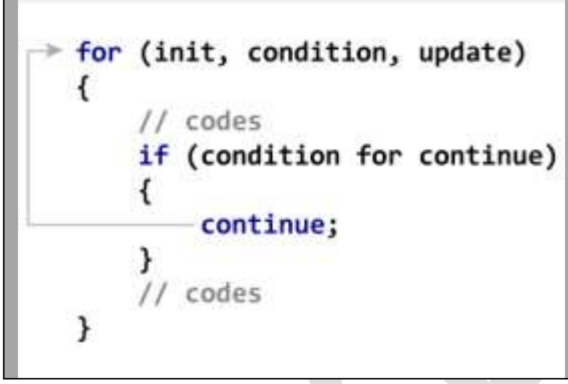
Continue

- During looping, sometimes it is necessary to skip a part of the loop under some condition and continue with the rest of the loop.
- This can be done using **continue** statement.
- A **continue** statement causes a loop to continue in the next iteration after skipping any statement after **continue.**

Syntax:



```
while (test Expression)
{
     // codes
     if (condition for continue)
     {
          continue;
     }
     // codes
}
```



```
for (init, condition, update)
{
     // codes
     if (condition for continue)
     {
          continue;
     }
     // codes
}
```

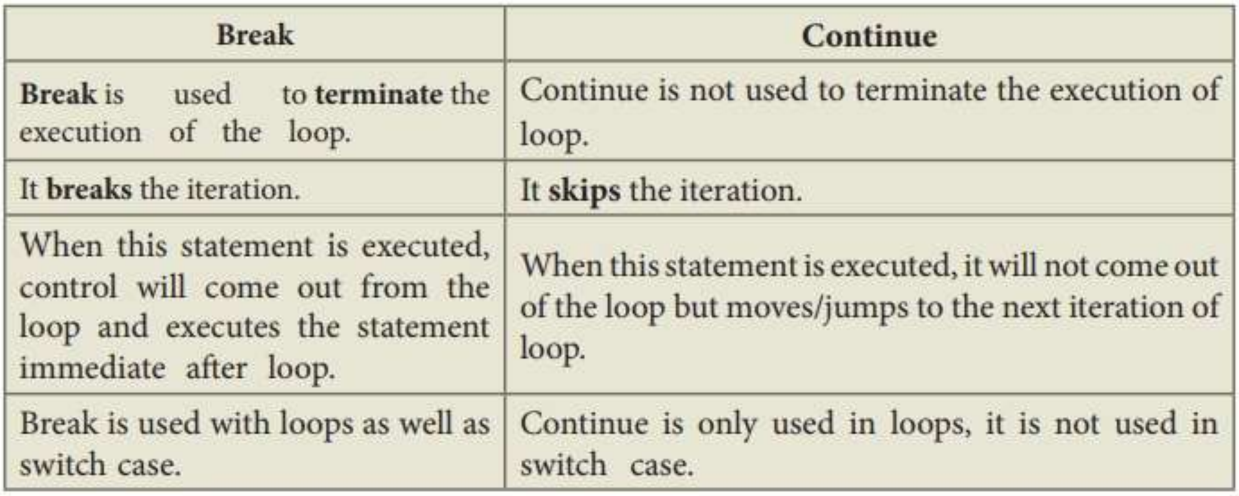
Program to display of 1 to N no's skipping 5th no.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i, n=20;
   clrscr();
   for(i=1;i<=n;++i)
     {
          if(i % 5 == 0)
         {
                      continue;
         }
      printf("%d\n",i);
   }
   getch();
}
```

The above program will print all numbers other than multiples of 5. i.e. 1 , 2 , 3, 4, 6, 7, 8, 9, 11, etc. It will skip every 5th number and the loop will continue normally again.

Note:

- We can also jump out of a program using **exit** function.
- It is available in **stdlib.h** header file.
- It specify 0 as argument terminate a program normally.
- Non-zero value means that a program will terminate due to some error condition.



Break	Continue
Break is used to **terminate** the execution of the loop.	Continue is not used to terminate the execution of loop.
It **breaks** the iteration.	It **skips** the iteration.
When this statement is executed, control will come out from the loop and executes the statement immediate after loop.	When this statement is executed, it will not come out of the loop but moves/jumps to the next iteration of loop.
Break is used with loops as well as switch case.	Continue is only used in loops, it is not used in switch case.

Unit 5: Concepts of Arrays and Pointers

Array

"An array is a fixed-sized sequential collection of elements having same datatype."

Why arrays are used?

- We have different datatypes like **int, float, etc** to declare variables but a variable can hold only a single value at a time.
- Therefore, to store a large quantity of numbers, we need multiple variables.
- For multiple variables we need to remember multiple names which makes a program complex and difficult to understand.
- Therefore, large numbers can be stored in a single name and can be accessed efficiently by using arrays.

5.1 Concept of One-Dimensional Array (Single-subscripted variable)

5.1.1 Numeric 1-D Array

- A list of elements can be given a single variable name using a single size which is known as a subscript and is specified during the array variable declaration in square brackets.
- Such a variable is called Single-subscripted variable or 1-D Array.

Declaration of 1-D Array

- Like other variables, arrays also must be declared before they are used in a program so that the compiler can allocate required space for them in memory.
- We can declare an array using the following syntax:

  Data-type var-name[size]; E.g. int a[10];

40	55	63	17	22	68	89	97	89	
0	1	2	3	4	5	6	7	8	<- **Array Indices**

Array Length = 9
First Index = 0
Last Index = 8

- The datatype can be int, float, char, double and size is an integer value that specifies the maximum number of elements that can be stored inside the array. [size cannot be negative]
- Here **a** is an array of 9 elements of integer type.

Note:

- Reference to arrays outside the size limits will not give any error.
- **char** name[10]; Here name is an array of characters and is known as a string variable.

Initializing an Array

- After an array is declared it must be initialized. Otherwise, it will contain garbage value (any random value).
- An array can be initialized at either compile time or at runtime.

Compile Time: **type array_name[size] = { list_of_values };**

```
E.g.    int marks[4]={57,67,87,77};

        float area[3]={ 23.4, 6.8, 5.5 };

        char name[6]={ 'H', 'E', 'L', 'L' };
```

Note:

- If the numbers of values are less than the size of array, the remaining else are initialized to zero if the array is numeric and to '\0' if the array is character.
- **No Index Out of bound Checking:** There is no index out of bound checking in C.

```
E.g.        int a[3];
            printf("%d", a[-2]); will run fine
```

- In C, it is a compiler error to initialize an array with more elements than specified size.

```
E.g.        int a[3] = { 10, 20, 30 ,40 };          // runs fine in C++ but not in C
```

Accessing an array

- To access the elements of array we use index. Index specifies the position of the element in the array.
- **Index** of an array starts from 0 to size-1 i.e first element of **a** array will be stored at **a[0]** address and the last element at **a[9]** for an array of 10 elements **int a[10]**;

```
    E.g.    int a[4]={ 67, 87, 56, 77 };
            for(i=0; i<4; i++)
            {
                    printf("\n%d" , a[i]);
            }
```

Runtime:

- An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values.

E.g. : Write a Program to calculate sum of 10 elements taken from the user

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[10], sum=0;
   int i, j;
   printf("\nInput array elements:");
for(i=0; i<10; i++)
   {
          Printf("\nEnter A[%d] : -", i);
          scanf("%d",&a[i]);                    //Run
time array initialization
          sum = sum + a[i];
   }
     printf("\n Sum of elements is :- %d", sum);
   getch();
}
```

5.1.2 Numeric 1-D Array Operations

Operations on Arrays

- An array is a collection of items which can be referred to by a single name.
- An array is also called a linear data structure because the array elements lie in the computer memory in a linear fashion.
- The possible operations on array are:
  - Insertion
  - Deletion
  - Traversal
  - Searching
  - Sorting
  - Merging
  - Updating

Insertion and Deletion

- Inserting an element at the end of the linear array can be easily performed, provided the memory space is available to accommodate the additional element.
- If we want to insert an element in the middle of the array then it is required that half of the elements must be moved rightwards to new locations to accommodate the new element and keep the order of the other elements.
- Similarly, if we want to delete the middle element of the linear array, then each subsequent element must be moved one location ahead of the previous position of the element.

Traversing

- Traversing basically means the accessing the each and every element of the array at least once.
- Traversing is usually done to be aware of the data elements which are present in the array.

Merging

- Merging is the process of combining two arrays in a third array. The third array must have to be large enough to hold the elements of both the arrays. We can merge the two arrays after sorting them individually or merge them first and then sort them as the user needs.

Updating

- The updating of elements in an array can be done by either specifying a particular element to be replaced or by identifying a position where the replacement has to be done.
- For updating, we generally require the following details.
  1. Elements of an array
  2. Position/element, where it has to be inserted
  3. The value to be inserted.

5.1.2.1 Sorting Array in Ascending or Descending Order

- Sorting is the process of arranging elements in ascending or descending order.
- A sorted list is called ordered list. It is helpful in doing rapid searching operations.
- There are many sorting techniques available like:
  1. **Bubble Sort**
  2. **Selection Sort, etc**

Bubble Sort

- Bubble sort is also known as sinking sort.
- This algorithm compares each pair of adjacent items (next Item) and swaps them if they are in the wrong order, and this same process goes on until no swaps are needed.

Steps of Bubble Sort
Let’s Assume Initially Array as A(12, 3, 5, 2, 1)

12	**3**	**5**	**2**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

Pass 1:

- Start at index zero, compare the element with the next one (a[0] & a[1], and swap if a[0] > a[1]. Here A[0] > A[1] => 12 > 3 True. Therefore will swap A[0] and A[1]. So the new Array is:

3	**12**	**5**	**2**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Now compare a[1] & a[2] and swap if a[1] > a[2]. Here A[1] > A[2] => 12 > 5 True. Therefore So the new Array is:

3	**5**	**12**	**2**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- In the next step, we compare A[2] and A[3]. 12 > 2. Therefore new array is:

3	**5**	**2**	**12**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- In the next step, we compare A[3] and A[4]. 12 > 1. Therefore new array is:

3	**5**	**2**	**1**	**12**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Repeat this process until the end of the array. After doing this, the largest element is present at the end. This whole thing is known as a **pass**. In the first pass, we process array elements from [0,n-1].
- Repeat step one but process array elements [0, n-2] because the last one, i.e., a[n-1], is present at its correct position. After this step, the largest two elements are present at the end.

Pass 2:

- In the next step, we compare A[0] and A[1]. 3 > 5 False. Therefore new array is:

3	**5**	**2**	**1**	**12**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Repeat this process n-1 times.

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

void main()
{
        int a[10],i, j,n, t;
        clrscr();

        printf("\nEnter N:-");
        scanf("%d",&n);

        for(i=0; i < n; i++)
        {
                printf("\nEnter A[%d]:-",i);
                scanf("%d",&a[i]);
        }

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

for(i=0; i < n; i++)
        {
                for(j=0; j<n-i-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                t = a[j];
                                a[j] = a[j+1];
                                a[j+1] = t;
                        }
                }
        }


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


getch();
}
```

Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
    1. The subarray which is already sorted.
    2. Remaining subarray which is unsorted.
- In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Steps of Selection Sort
Let's Assume Initially Array as A(12, 3, 5, 2, 1)

12	**3**	**5**	**2**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

Pass 1:

- In step 1, Compare A[0] & A[1]. Here A[0] > A[1] => 12 > 3. Therefore swap them. The new array is

3	**12**	**5**	**2**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Next Compare A[0] & A[2]. Here A[0] > A[2] => 3 > 5? False. Therefore no swapping.
- Next Compare A[0] & A[3]. Here A[0] > A[3] => 3 > 2. Therefore swap them. The new array is

2	**12**	**5**	**3**	**1**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Next Compare A[0] & A[4]. Here A[0] > A[4] => 2 > 1. Therefore swap them. The new array is

1	**12**	**5**	**3**	**2**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- After doing this, the smallest element is present at the first position. This whole thing is known as a **pass**. In the first pass, we process array elements from [0,n-1].
- Repeat step one but process array elements [1, n-1] because the first one, i.e., a[0], is present at its correct position. After this step, the smallest two elements are present at the first.

Pass 2:

- Now Compare A[1] & A[2]. Here A[1] > A[2] => 12 > 5. Therefore swapping them. The new array is

1	**5**	**12**	**3**	**2**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- Now Compare A[1] & A[3]. Here A[1] > A[3] => 5 > 3. Therefore swapping them. The new array is

1	**3**	**12**	**5**	**2**
A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**

- This process continues till [n-2, n-1].

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

void main()
{
        int a[10],i, j,n, t;
        clrscr();

        printf("\nEnter N:-");
        scanf("%d",&n);

        for(i=0; i < n; i++)
        {
                printf("\nEnter A[%d]:-",i);
                scanf("%d",&a[i]);
        }

        printf("\nBefore Sorting\n");
        for(i=0; i < n; i++)
        {
                printf("\n%d",a[i]);
        }
for(i=0; i < n; i++)
        {
                for(j=i+1; j<n;j++)
                {
                        if(a[i]>a[j])
                        {
                                t = a[i];
                                a[i] = a[j];
                                a[j] = t;
                        }
                }
        }


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


getch();
}
```

5.1.2.2 Searching Element from an Array (Linear Search)

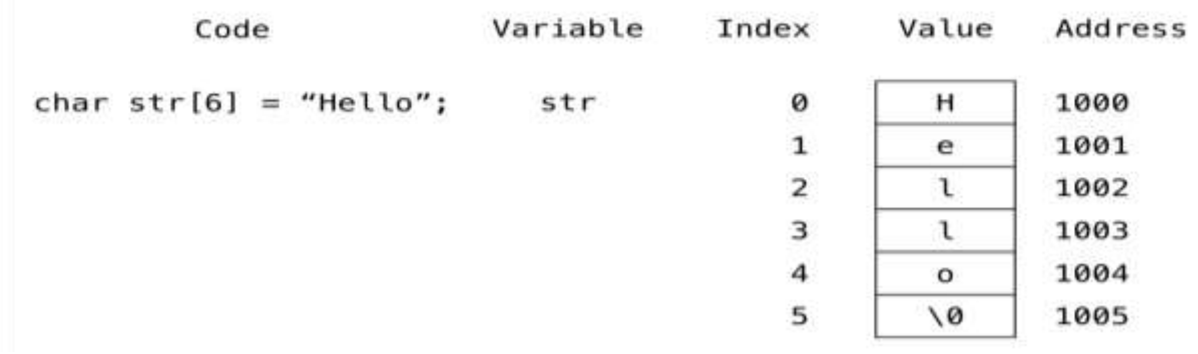
- Searching an element in an array, the search starts from the first element till the last element.
- The average number of comparisons in a sequential search is (N+1)/2 where N is the size of the array.
- If the element is in the 1st position, the number of comparisons will be 1 and if the element is in the last position, the number of comparisons will be N.

- We begin searching the element N in the Array. For every pass of the loop we compare N with the current Element i.e. A[0] == N?, A[1] == N? And so on till we find the correct match.

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

void main()
{
        int a[10],i, j,n, t;
        clrscr();

        printf("\nEnter N to search:-");
        scanf("%d",&n);

        for(i=0; i < 5; i++)
        {
                printf("\nEnter A[%d]:-",i);
                scanf("%d",&a[i]);
        }
for(i=0; i < 5; i++)
        {
                If(a[i] == n)
                {
        Printf("\n%d is found at %d position", n , i+1);
                }
        }

getch();
}
```

5.1.3 Character 1-D Array

- The C language treats strings as arrays of characters.
- The compiler terminates the character strings with an additional null ('\0') character.
- A String is a one dimensional array of characters in c.
- The element name[10] holds the null character '\0' at the end.
- When declaring character arrays, we must always allow one extra element's space for the null terminator '\0'.
- A 1-D Character Array can be declared in the same way as 1-D int array.



Code	Variable	Index	Value	Address
char str[6] = "Hello";	str	0	H	1000
		1	e	1001
		2	l	1002
		3	l	1003
		4	o	1004
		5	\0	1005

Declaration and Initialization of Character Array

- There are different ways to initialize a character array variable.
  - char name[6]="Hello"; //valid initialization
  - char name[ ]={'H','E','L','L','O','\0'}; // valid Init
  - char name[10]={'H','E','L','L','O','\0','\0','\0','\0','\0'}; // Here the remaining space will be initialized with null characters.

- Some example for illegal Initialization of character array are,
  1. char ch[3]="hello"; // Illegal (Size is less than no of chars)
  2. char str[4];
str = "hello"; //illegal (We cannot directly assign a string with =)

Reading a 1-D Character Array (Run-time)

- We can read a 1-D char array from the user in the following way:
  1. Using getchar() [discussed in earlier chapter]
  2. Using scanf

     ```
     char str[10];
     scanf("%s",str);
     ```
  3. Using gets()

     ```
     char str[10];
     gets(str);
     ```

5.1.3.1 Character Array Operations

- Strings can be reversed, compared, joined, etc.
- We can perform the following operations on 1-D character Array or on a string by using the functions in the <string.h> header file.
  1. Reverse a char array (**strrev()**)
  2. Comparing two char array (**strcmp()**)
  3. Joining two char arrays (**strcat()**)
  4. Copying a char array in another (**strcpy()**)
  5. Calculating the length of char array (**strlen()**)

5.1.3.2 Use of \0, \n, \t

- **\n :** It is used in the output statement to give the output on a new line.

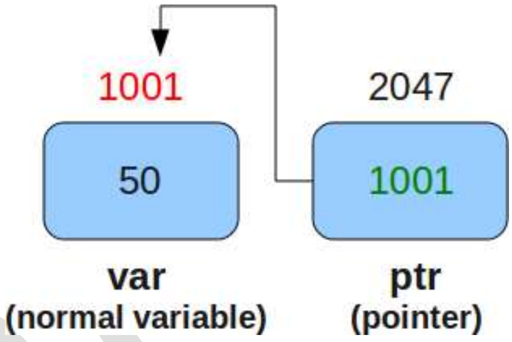
  Printf("Good \n Morning "); // O/p: Good
Morning
- **\t :** It is used in the output statement to give the output with a horizontal tab.

  Printf("Good \n Morning "); // O/p: Good Morning
- **\0 :** It is used as a String Terminator The String is a variable length data structure and is stored in a fixed-length array.
- The size of the array is not always the size of the string and most often it is much larger than the string stored in it.
- Therefore, the last element of the array need not represent the end of the string.
- We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

5.2 Pointers

5.2.1 Concept of Pointers

- A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- It is a derived data type.
- Let us assume that system has allocated memory location 1001 for a variable **int var = 50 ;**
- Since the memory addresses are simply numbers they can be assigned to some other variable.
- The variable that holds memory address are called **pointer variables (Here ptr).**
- A pointer variable is therefore nothing but a variable that contains an address, which is a location of another variable.



- Value of pointer variable will be stored in another memory location. Here **2047.**

Benefit of using pointers

- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length and the program execution time.
- It allows C to support dynamic memory management.
- They can make some things much easier, help improve our program's efficiency, and even allow us to handle unlimited amounts of data.

5.2.2 Declaring and initializing int, float, char, and void pointers

- Like any variable or constant, we must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
Data-type *var-name;
```

- Here, data-type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable.
- The asterisk unary operator * is used to declare a pointer that returns the value of the variable located at the address specified by its operand. It is also known as Indirection.
- **Pointer Initialization** is the process of assigning address of a variable to pointer variable.
- Pointer variable contains address of variable of same data type.
- Ptr can be assigned the address of **a** using address operator (&).

```
int a = 10, *ptr;
ptr = &a;
```

- Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using **the indirection operator** *or **deferencing operator**.
- Accessing the value of **a** using ptr variable → printf("%d",*ptr) // Displays a
- We can also declare pointers of char, float and void type. Initializing float and double pointers will be same as int.

  E.g. float x; double y;

char *str;   float *ptr = &x;   double *ptr1 = &y;

Char pointer

- Initializing char pointer is slightly different than others. We can initialize char ptr as follows:
  char *str = “Hello”;   or   char *str = “”; // Empty String
- **Character pointers** allow us to declare and store strings of variable length i.e. the length is not fixed.

Void Pointer

- A **void pointer** is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.
  E.g.
  ```
  int a = 10;
  char b = 'x';
  void *p = &a; // void pointer holds address of int 'a'
  p = &b; // void pointer holds address of char 'b'
  ```
- void pointers cannot be dereferenced. i.e. **printf(“%d”,*p);**   // is invalid
- But they can be cast to any type i.e. **printf(“%d”, *(int *)p);**   // is perfectly valid

5.2.3 Pointer to 1-Dimensional Numeric Array

- In C, pointers and arrays are very closely related. We can access the elements of the array using a pointer.
- The compiler also access elements of the array using pointer notation rather than subscript notation because accessing elements using pointer is very efficient as compared to subscript notation.
- The name of the array is a constant pointer that points to the address of the first element of the array or the base address of the array.
- We can declare and assign a pointer to a 1-D array as follows:
  ```
  int *ptr, a[5] = {1, 2, 3, 4, 5};
  ptr = a;          // ptr = &a[0];
  ```
- Now to access the elements of array using ptr we can use *ptr instead of a[0].
  ```
  printf(“\n%d”,*ptr);  // Will print 1
  ptr++;                // Jumps to next element
  printf(“\n%d”,*ptr);  // Now it will print 2
  ```

Program to Access 1-D array using Pointer

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

        for(i=0; i<10; i++)
        {
                printf("\nEnter A[%d]:-",i);
                scanf("%d",&a[i]);
        }
        ptr = &a[0];    // ptr = a;
printf("\nArray is:-\n");


        while(ptr != &a[9])
        {
                printf("\n%d",*ptr);
                ptr = ptr + 1; // ptr++;
        }
getch();
}
```