UNIT -I

PROBLEM SOLVING THROUGH PROGRAMMING

Chapter 1: Introduction to Computer System

INTRODUCTION TO COMPUTERS AND PROBLEM SOLVING

Today, almost all of us in the world make use of computers in one way or the other. It finds applications in various fields of engineering, medicine, commercial, research and others. Not only in these sophisticated areas, but also in our daily lives, computers have become indispensable. They are present everywhere, in all the devices that we use daily like cars, games, washing machines, microwaves etc. and in day to day computations like banking, reservations, electronic mails, internet and many more.

The word **computer** is derived from the word **compute.** Compute means to calculate. The computer was originally defined as a superfast calculator. It had the capacity to solve complex arithmetic and scientific problems at very high speed. But nowadays in addition to handling complex arithmetic computations, computers perform many other tasks like accepting, sorting, selecting, moving, comparing various types of information. They also perform arithmetic and logical operations on alphabetic, numeric and other types of information. This information provided by the user to the computer is **data.**

The information in one form which is presented to the computer is the input information or **input data.** Information in another form is presented by the computer after performing a process on it. This information is the output information or **output data.** The set of instructions given to the computer to perform various operations is called as the **computer program.** The process of converting the input data into the required output form with the help of the computer program is called as **data processing.** The computers are therefore also referred to as data processors**.** Therefore a computer can now be defined as a fast and accurate data processing system that accepts data, performs various operations on the data, has the capability to store the data and produce the results on the basis of detailed step by step instructions given to it.

The terms **hardware and software** are almost always used in connection with the computer.

The Hardware:

The hardware is the machinery itself. It is made up of the physical parts or devices of the computer system like the electronic Integrated Circuits (ICs), magnetic storage media and other mechanical devices like input devices, output devices etc. All these various hardware are linked together to form an effective functional unit. The various types of hardware used in the computers, has evolved from vacuum tubes of the first generation to Ultra Large Scale Integrated Circuits of the present generation.

The Software:
The computer hardware itself is not capable of doing anything on its own. It has to be given explicit instructions to perform the specific task. The computer program is the one which controls the processing activities of the computer. The computer thus functions according to the instructions written in the program. Software mainly consists of these computer programs, procedures and other documentation used in the operation of a computer system. Software is a collection of programs which utilize and enhance the capability of the hardware.

CLASSIFICATION OF COMPUTERS:

Computers are broadly classified into two categories depending upon the logic used in their design as:

1. **Analog computers:**
   In analog computers, data is recognized as a continuous measurement of a physical property like voltage, speed, pressure etc. Readings on a dial or graphs are obtained as the output, ex. Voltage, temperature; pressure can be measured in this way.

2. **Digital Computers:**

   These are high speed electronic devices. These devices are programmable. They process data by way of mathematical calculations, comparison, sorting etc. They accept input and produce output as discrete signals representing high(on) or low (off) voltage state of electricity. Numbers, alphabets, symbols are all represented as a series of 1s and Os.

   Digital Computers are further classified as General Purpose, Digital Computers and Special Purpose Digital Computers. General Purpose computer can be used for any applications like accounts, payroll, data processing etc. Special purpose computers are used for a specific job like those used in automobiles, microwaves etc.

Another classification of digital computers is done on the basis of their capacity to access memory and size like:

- **Microcomputers:** Microcomputers are generally referred to as **Personal Computers (PCs).** They have smallest memory and less power. They are widely used in day to day applications like office automation, and professional applications, ex. PCAT, Pentium etc.

- **Note Book and Laptop Computers:** These are portable in nature and are battery operated. Storage devices like CDs, floppies etc. and output devices like printer scanners can be connected to these computers. Notebook computers are smaller in physical size than laptop computers. However, both have powerful processors, support graphics, and can accept mouse driven input.

- **Hand Held Computers:**
  These types of computers are mainly used in applications like collection of field data. They are even smaller than the note book computers.

- **Hybrid Computers:** Hybrid Computers are a combination of Analog and Digital computers. They combine the speed of analog computers and accuracy of digital computers. They are mostly used in specialized applications where the input data is in an analog form i.e. measurement. This is converted into digital form for further processing. The computers accept data from sensors and produce output using conventional input/output devices.

- **Mini Computers:** Mini computers are more powerful than the micro computers. They have higher memory capacity and more storage capacity with higher speeds. These computers are mainly used in process control systems. They are mainly used in applications like payrolls, financial accounting, Computer aided design etc. ex. VAX, PDP-11.

- **Mainframe Computers:** Main frame computers are very large computers which process data at very high speeds of the order of several million instructions per second. They can be linked into a network with smaller computers, micro computers and with each other.

They are typically used in large organizations, government departments etc. ex. IBM4381, CDC.

- **Super Computers:** A super computer is the fastest, most powerful and most expensive computer which is used for complex tasks that require a lot of computational power. Super computers have multiple processors which process multiple instructions at the same time. This is known as **parallel processing.** These computers are widely used in very advanced applications like weather forecasting, processing geological data etc. ex. CRAY-2, NEC - 500, PARAM.

PROBLEM SOLVING ASPECT

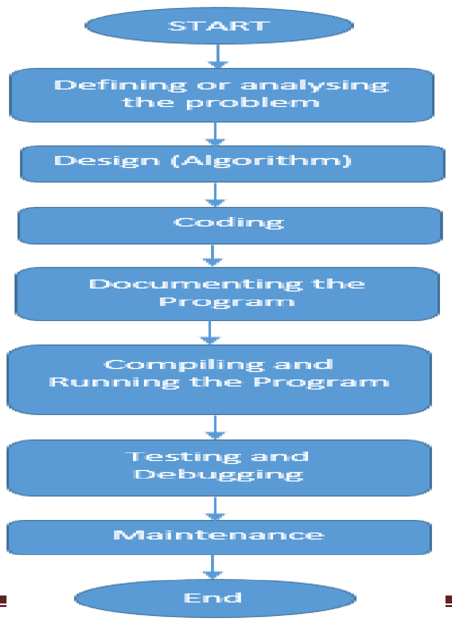
PROGRAM DEVELOPMENT STEPS

Problem solving is a creative process. It is an act of defining a problem, determining the cause of the problem, identifying, prioritizing, and selecting alternatives for a solution and implementing a solution.

It is important to see if there are any similarities between the current problem and other problems that have already been solved. We have to be sure that the past experience does not hinder us in developing new methodology or technique for solving a problem. The important aspect to be considered in problem-solving is the ability to view a problem from a variety of angles.

The various steps involved in Program Development are:

- Defining or Analyzing the problem
- Design (Algorithm)
- Coding
- Documenting the program
- Compiling and Running the Program
- Testing and Debugging
- Maintenance



Defining or Analyzing the Problem:

In general terms, this step entails

- Identifying the desired results (output),
- Determining what information (input) is needed to produce these results,
- Figuring out what must be done to proceed from the known data to the desired output (processing).

Although this step is described in one sentence, actually it may be the hardest part. And it is certainly the most important part! When you analyze the problem, you determine what the result will be. If you don’t do this correctly, all the elegant code in the world, written and executed flawlessly, will not solve your problem.

Design (Algorithm):

To design a program means to create a detailed description, using relatively ordinary language or special diagrams of the program to be created. Typically, this description is created in stages, proceeding from simple to more complex, and consists of a number of step by-step procedures (algorithms) that combine to solve the given problem.
An algorithm is like a recipe. It is a step-by-step method for solving a problem or doing a task. Algorithms abound in programming, mathematics, and sciences and are common in everyday life as well. For example, you are making use of an algorithm when you follow a recipe to bake a cake or go through the process of using an ATM machine.
Therefore, an algorithm must contain

- clear,
- unambiguous,
- step-by-step instructions.
- No step, not even the most basic and elementary, can be left out.

When we design a program, we do not completely define the algorithms all at once. As we develop the program and break down major components into smaller pieces, the algorithms for each task become more complex and detailed.

Coding

Once you have designed a suitable program to solve a given problem, you must translate that design into program code; that is, you must write statements (instructions) in a particular programming language such as C++, Visual Basic, or JavaScript etc. to put the design into a usable form.

Additional statements are included at this point to document the program. Documentation is a way to provide additional explanation in plain English (or other mother tongue) that the computer ignores but which makes it easier for others to understand the program code. Normally, a programmer provides internal and external documentation. Internal documentation exists within the code and explains it. External documentation is provided separate from the program in a user's guide or maintenance manual.

The ways that specific words and symbols are used by each language is called its *syntax* (the rules that govern the structure of the language).

Documenting the program

Documentation explains how the program works and how to use the program. Documentation can be of great value, not only to those involved in maintaining or modifying a program, but also to the programmers themselves. Details of particular programs, or particular pieces of programs, are easily forgotten or confused without suitable documentation.
Documentation comes in two forms:

- External documentation, which includes things such as reference manuals, algorithm descriptions, flowcharts, and project workbooks
- Internal documentation, which is part of the source code itself (essentially, the declarations, statements, and comments)

Compiling and Running the Program

Compilation is a process of translating a source program into machine understandable form. The compiler is system software, which does the translation after examining each instruction for its correctness. The translation results in the creation of object code.

After compilation, Linking is done if necessary. Linking is the process of putting together all the external references (other program files and functions) that are required by the program. The program is now ready for execution. During execution, the executable object code is loaded into the computer's memory and the program instructions are executed.

Testing and Debugging

Testing is the process of executing a program with the deliberate intent of finding errors. Testing is needed to check whether the expected output matches the actual output. Program should be tested with all possible input data and control conditions. Testing is done during every phase of program development. Initially, requirements can be tested for its correctness. Then, the design (algorithm, flow charts) can be tested for its exactness and efficiency.

Debugging is a process of correcting the errors. Programs may have logical errors which cannot be caught during compilation. Debugging is the process of identifying their root causes. One of the ways to ensure the correctness of the program is by printing out the intermediate results at strategic points of computation.

Some programmers use the terms "testing" and "debugging" interchangeably, but careful programmers distinguish between the two activities. Testing means detecting errors. Debugging means diagnosing and correcting the root causes. On some projects, debugging occupies as much as 50 percent of the total development time. For many programmers, debugging is the hardest part of programming because of improper documentation.

Maintenance

Programs require a continuing process of maintenance and modification to keep pace with changing requirements and implementation technologies. Maintainability and modifiability are essential characteristics of every program. Maintainability of the program is achieved by:

- Modularizing it
- Providing proper documentation for it
- Following standards and conventions (naming conventions, using symbolic constants etc.)

INTRODUCTION TO PROGRAMMING LANGUAGES

What is a Programming Language?
Computer Programming is an art of making a computer to do the required operations, by means of issuing sequence of commands to it.

A programming language can be defined as a vocabulary and set of grammatical rules for instructing the computer to perform specific tasks. Each programming language has a unique set of characters, keywords and the syntax for organizing programming instructions.
The term programming languages usually refers to high-level languages, such as BASIC, C, C++,COBOL, FORTRAN, Ada, and Pascal.

Why Study Programming Languages?
The design of new programming languages and implementation methods have been evolved and improved to meet the change in requirements. Thus, there are many new languages.
The study of more than one programming language helps us:

# to master different programming paradigms
# to enhance the skills to state different programming concepts
# to understand the significance of a particular language implementation

# to compare different languages and to choose appropriate language
# to improve the ability to learn new languages and to design new languages

TYPES AND CATEGORIES OF PROGRAMMING LANGUAGES:

Types of Programming Languages

There are two major types of programming languages:
# Low Level Languages
# High Level Languages

Low Level Languages
The term low level refers closeness to the way in which the machine has been built. Low level Languages are machine oriented and require extensive knowledge of computer hardware architecture and its configuration. Low Level languages are further divided in to **Machine language and Assembly language.**

(a) Machine Language
Machine Language is the only language that is directly understood by the computer. It does not need any translator program. The instructions are called machine instruction (machine code) and it is written as strings of 1's (one) and 0's (zero). When this sequence of codes is fed in to the computer, it recognizes the code and converts it in to electrical signals.

For example, a program instruction may look like this: 1011000111101
Machine language is considered to be the first generation language. Because of its design, machine language is not an easy language to learn. It is also difficult to debug the program written in this language.

Advantage
#The program runs faster because no translation is needed. (It is already in machine understandable form)

Disadvantages
# It is very difficult to write programs in machine language. The programmer has to know details of hardware to write program
# It is difficult to debug the program

(b) Assembly Language

In assembly language, set of mnemonics (symbolic keywords) are used to represent machine codes. Mnemonics are usually combination of words like ADD, SUB and LOAD etc. In order to execute the programs written in assembly language, a translator program is required to translate it to the machine language. This translator program is called Assembler. Assembly language is considered to be the second-generation language.

Advantages:

# The symbolic keywords are easier to code and saves time and effort
# It is easier to correct errors and modify programming instructions
# Assembly Language has utmost the same efficiency of execution as the machine level language, because there is one-to-one translation between assembly language program and its corresponding machine language program

Disadvantages:

#Assembly languages are machine dependent. A program written for one computer
might not run in other computer.

High Level Languages

High level languages are the simple languages that use English like instructions and mathematical symbols like +, -, %, /, for its program construction. In high level languages, it is enough to know the logic and required instructions for a given problem, irrespective of the type of computer used. Compiler is a translator program which converts a program in high level language in to machine language. Higher level languages are problem-oriented languages because the instructions are suitable for solving a particular problem.

For example, COBOL (Common Business Oriented Language) is mostly suitable for business Oriented applications. There are some numerical & mathematical oriented languages like FORTRAN (Formula Translation) and BASIC (Beginners All-purpose Symbolic Instruction Code).

Advantages of High Level Languages

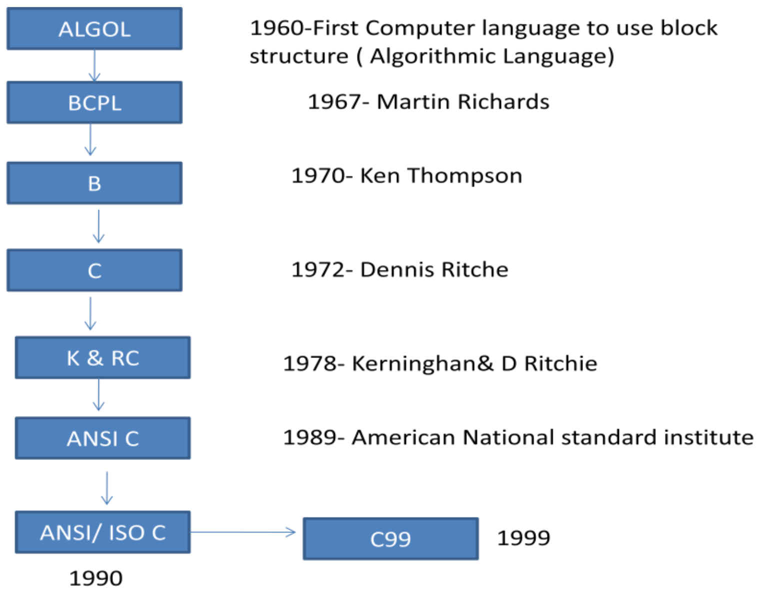
#High level languages are easy to learn and use

What makes a Good Language?
Every language has its strengths and weaknesses.
For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organize large programs.
PASCAL is very good for writing well structured and readable programs, but it is not as flexible as the C programming language.

EVOLUTION & CHARACTERISTICS OF C LANGUAGE

During second generation of Computers (1956-57), commercial application problems used computers and this further increased in mid 60s.

- This lead to the development of a series of High-Level programming languages like BASIC, FORTRAN, COBOL, ALGOL, PASCAL, SNOBAL, PROLOG, LISP, etc., which were effectively used in problem solving.
- These High-Level programming languages were application specific
  Ex. FORTRAN was widely used for scientific processing
  COBOL was used for business data processing
  PASCAL was used for general application
  PROLOG & LISP for artificial intelligence applications
- Nowadays high level languages like C, C++, C# are becoming increasingly popular and earlier programming languages like BASIC, FORTRAN, COBOL, PASCAL, etc. are becoming outdated.
- This is because C language being a high level language satisfies varying requirements of users - like
  It can be used as systems programming language.
  The UNIX OS has been written in C.

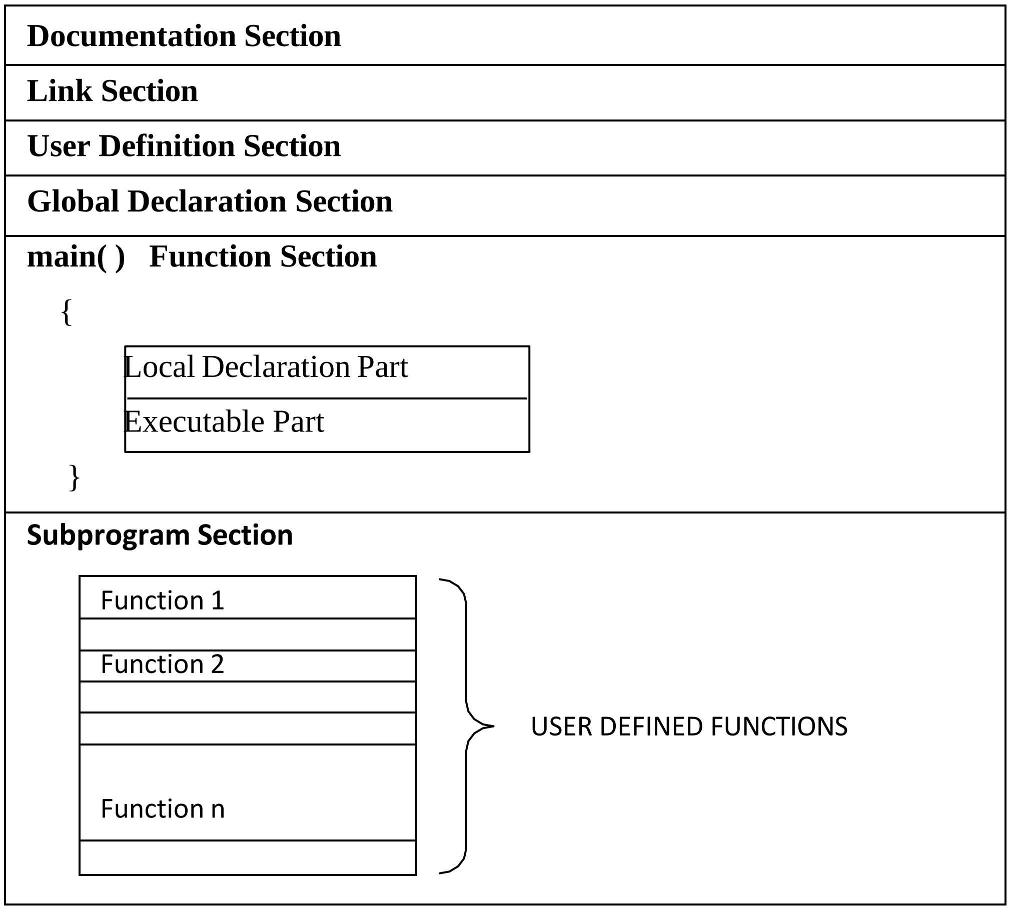

ALGOL
1960-First Computer language to use block structure (Algorithmic Language)
BCPL
1967- Martin Richards
B
1970- Ken Thompson
C
1972- Dennis Ritche
K & RC
1978- Kerninghan& D Ritchie
ANSI C
1989- American National standard institute
ANSI/ ISO C
1990
C99
1999

- Algol appeared only few years after FORTRAN, but was much more sophisticated.
- CPL was also big, but had very large number of features which made it difficult to learn and implement it.
- BCPL was modified CPL including only basic features.
- B was written by Ken Thompson for an early implementation of UNIX, which was a further simplification of CPL.
  (Both BCPL and B were useful only while dealing with certain kind of problems)
- Ritchie's achievement in C was to restore some of the lost generality in BCPL & B, mainly by the cunning USE OF DATA TYPES.
- In 1978, Dennis Ritchie and Brian Kernighan jointly published a detailed description of C language document, known as K & R 'C'. It was extensively used in Bell labs before it was released to commercial applications in 1978.
- Some of the drawbacks of K & R 'C' implementations are overcome by ANSI (American National Standards Institute) standards.

Characteristics of ‘C’

- ‘C’ is a **general purpose programming language**.
  1. It can be used for systems programming, and
  2. It can be used to develop application programs required to solve a variety of scientific and engineering problems.
- ‘C’ language offers a close interaction with the interior (hardware) of the computer. It can be used to configure or change the hardware set-ups, memory locations/register contents, etc. Hence ‘C’ may be called a **Middle Level language** as it can be used both as a High Level language and a Low Level language.
- ‘C’ is a **structured programming language**. Program can be structured in form of functional modules or blocks. A proper collection of these modules makes a complete program. Such modular structure makes program debugging, testing and maintenance easier.
- ‘C’ is **highly portable**. That is, programs written on one computer can be run on any other computer with little or no modifications.
- ‘C’ has a **rich set of built-in-functions and operators** which can be used to write any complex program.
- ‘C’ supports **various data types** like integer numbers, floating point numbers, characters, etc.
- ‘C’ has a very **few keywords** (reserved words – only 32).
- ‘C’ is **case sensitive**, for example **num** is different from **NUM**.
- ‘C’ **supports pointers** and operations on pointers.

BASIC STRUCTURE OF A ‘C’ PROGRAM



Documentation Section:

- Consists of a set of comment lines giving the name of the program, author, and other details which the programmer would like to use later.
- These are non executable lines i.e., compiler ignores these comment lines.

Example:

```
/*  Computing and Printing Employee’s Salary Slip  */
/*  Prepared and compiled at NMAMIT, NITTE  */
/*          Prepared on April, 2021              */
```

Link Section:

It provides instruction to the compiler to link functions from the system library.

Example :
```
#include <stdio.h>
#include <math.h>
```

#include <string.h>

User Definition Section:

In this section the user defines all SYMBOLIC CONSTANTS

```
Example :    #define PI 3.141
             #define CITY "Bangalore"
             #define CONDITION 'y'
             #define AVEARGE 76.35
```

Note: Link Section and User Definition Section are together called as the 'PREPROCESSOR STATEMENTS' OR 'PREPROCESSOR DIRECTIVES'

Global Declaration Section:

- Some variables or functions in the program are required to be accessed by more than one function. They have to be declared as global variables or global functions.
- Such Global declaration have to be made before the main( ) function.

main( ) Function:

- Every 'C' program must have only one main( ) function.
- Execution of 'C' program starts from the main( ) function.
- It should be written in lower case letters and should not be terminated by a semicolon.
- It calls other library functions and user defined functions

Braces:

- A pair of curly braces '{' & '}' are used in the main( ) function section.
- The left brace {represents the beginning of execution of the main function and the right brace} represents the end of execution of the main function.
- These braces are also used to indicate the begining and end of user-defined functions and compound statements.

Local Declaration Part:

- This part of the main( ) function declares all the variables used in the executable part
- Some of the variables may be initialized, i.e., assigned with initial values.

```
Examples:       int a,b=11,c;
                float m=23.05,n;
```

```
                    char c1='y',c2=' ',c3='n';
                    void Show(int x,int y);
```

Executable Part:

- This part consists of statements which are instructions to computer to perform specific operations.
- They may be input-output statements, arithmetic statements, control statements and other statements.
- All such statements terminate with a semi colon.

Sub Program Section:

- This section consists of user defined functions.
- User defined functions are functions written by user to perform a specific task.
- They are declared in the Global or Local declaration section.
- They may be written after or before the main( ) function.

```
/***********DOCUMENTATION SECTION*************/
/* PROGRAM TO COMPUTE AREA OF A CIRCLE
   PREPARED BY XYZ
   PREPARED ON 26-02-2006 */

/*************LINK SECTION*******************/
#include <stdio.h>
#include <conio.h>

/***********USER DEFINITION SECTION**********/
#define PI 3.141

/**********GLOBAL DEFINITION SECTION**********/
void Accept();
void Compute();
void Show();
float rad,area;

/**********MAIN() FUNCTION SECTION***********/
```

```
void main()
 {          /* Start of main() function         */
  clrscr();  /* To clear the dispaly screen       */
  Accept();  /* To read in the input data values */
  Compute(); /* To compute the result            */
  Show();    /* To display the result            */
 }          /* End of main() function           */

/***********SUB PROGRAM SECTION***************/

void Accept() /*User defined function 1 */
 {
  printf("\n\n\tEnter the radius of the circle:");
  scanf("%f",&rad);
  return; /*Return to main function */
 }

void Compute() /*User defined function 2 */
 {
  area=PI*rad*rad;
  return; /*Return to main function */}

void Show()  /*User defined function 3 */
 {
  printf("\n\n\tThe Area of circle is: %f",area);
  getch();
  return; /*Return to main function */
 }
```

C COMPILATION MODEL

Compiling and Executing a ‘C’ Program:

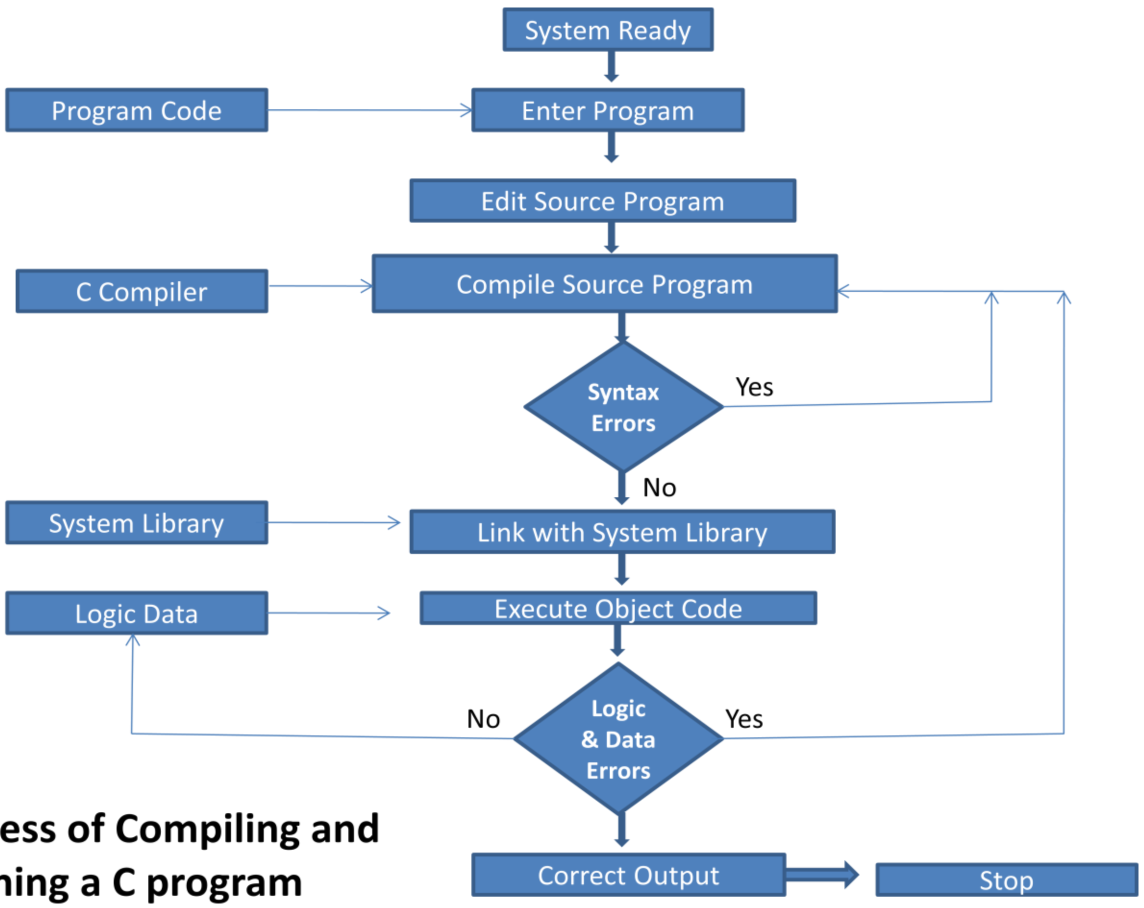
- Compiling a ‘C’ program means translating it into a computer understandable form known as machine language. Compilation is done by a ‘C’ compiler.

- A compiler is a program which accepts the source code (actual ‘C’ program) as input and translates it to machine understandable form.
- ‘C’ compilers are available with or without editors. An editor is a program which allows the programmer to type in (key in) the program and modify (edit) it.
- Integrated Development Environment (IDE): It is an environment (a package) where we find the compiler, editor, debugging tools, linking facilities, tracing and testing tools. Example: Code Blocks, Turbo C (TC), Borland C, Microsoft C/C++, ANSI C, etc.
- The procedures used in compiling and executing a ‘C’ program differ from one operating system to other.

Executing a C program written in C involves a series of steps, These are

- Creating the Program
- Compiling the Program
- Linking the program with the functions that are needed from the C library, and
- Executing the program.

Figure below illustrates the process of creating, compiling, and executing of C program, although these steps remains the same irrespective of the operating systems(OS) commands for implementing the steps and conventions for naming ***files*** may differ on different systems.
An OS is a program that controls the entire operation of computer system. All input/output operations are channeled through the OS. The OS which is an interface between the hardware and the user, handles the execution of user program.



Process of Compiling and Running a C program

‘C’ CHARACTER SET

Alphabets	Upper Case A to Z Lower Case a to z	
Digits	0 through 9	
Special Characters	, Comma . Period : Colon ; Semicolon ‘ Apostrophe (single quote) “ Double quote	/ Slash % Percentage & Ampersand ^ Caret ~ Tilde < Less than

	? Question mark ! Exclamatory mark _ Under score # Hash = Equal sign | Pipeline character + Plus sign – Minus sign * Asterisk	> Greater than \ Back slash (Left parenthesis) Right Parenthesis [Left bracket] Right bracket { Left brace } Right brace

<u>‘C’ TOKENS</u>

They are the smallest units or primitive elements of the grammar or syntax of ‘C’ language. Each and every line of a ‘C’ program will contain one or more ‘C’ tokens.

There are six types of tokens in ‘C’:

(1) Keywords (Reserved words)
(2) Identifiers (Variable names)
(3) Constants (Literals)
(4) Strings
(5) Operators
(6) Other special symbols

Example1: Consider the declaration statement in ‘C’

int n1, n2, sum ;

Here ‘C’ tokens are:	int	is a keyword
	n1 n2 sum	are identifiers/variables
	, , ;	are special symbols used as delimiters

Example2: Consider an assignment statement in ‘C’

bal = amount – debit ;

Here ‘C’ tokens are:

=, - , are operators

bal ,amount ,debit -> are identifiers/variables

; is a special symbol used as delimiters

KEYWORDS AND IDENTIFIERS

Keywords used in 'C':

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

Identifiers:

- They are names of variables representing data, labels, function names, array names, user-defined names, user-defined objects, etc.
- They are formed by sequences of alphabets and digits

Rules for forming identifier names:

(1) The first character must be an alphabet (uppercase or lowercase) or an underscore "_"
(2) All succeeding characters must be either letters or digits
(3) Uppercase and lowercase identifiers are different in 'C' i.e., identifiers are case sensitive
(4) No special characters or punctuation symbols are allowed except the underscore "_"
(5) No two successive underscores are allowed
(6) Keywords should not be used as identifiers

Examples of VALID identifiers in C

ping	sumup2	at_a_rate	story_of_1942
tick20	_check	balance	Q123

qlty_02	iNUM	Cost	one2next

Examples of INVALID identifiers in C

1stnum	+point	rate_in_$	flash@
min bal	Rs..	break	continue
tot__pay			

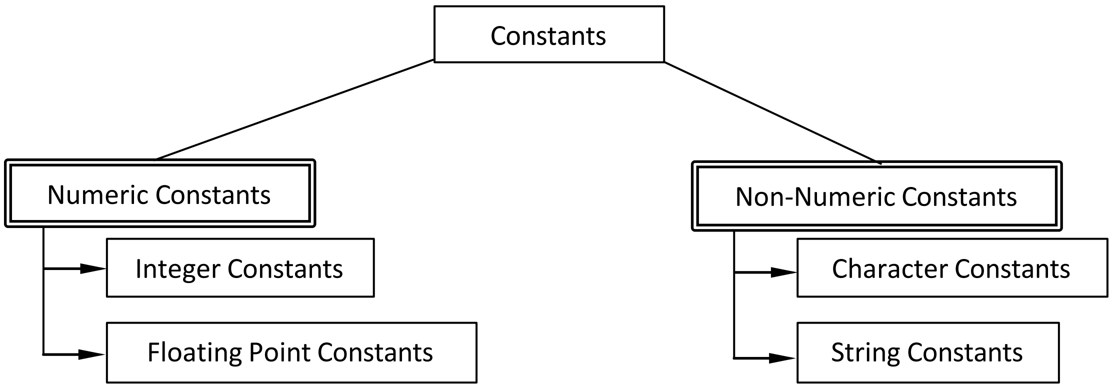
- Names of the identifiers once declared in C cannot be used to declare some other function name or array name in a single program.
- Number of significant characters to be recognized in an identifier varies from C compiler to C compiler

Constants and **Variables** are two essential ingredients for **Data Processing Applications** without which we may not be able to develop programs in any of the high level programming languages.

CONSTANTS

These are fixed values that do not change during the execution of a program. They are also known as **literals**.

In 'C' constants can be classified as:



Numeric Constants:

- They are simply numbers, a collection of one or more decimal digits.

- They are used for performing numeric calculations.

Non-Numeric Constant:

- These comprise of both numerals and letters of alphabets including special symbols.
- They are used in string manipulation applications like *Natural Language Processing (NPL)*

Integer Constants:

- Integer constants in ‘C’ represent whole numbers without any fractional part.
- It is a sequence of digits without a decimal point.
- It can be prefixed with a plus or minus sign.
- When minus sign doesn’t precede the number, it is assumed to be a positive one.
- There should not be any special symbols like blank spaces, comma, etc.

The general form of an integer constant is:

Sign	Digits

where,

Sign→ optional plus sign for positive numbers and minus sign for negative numbers.

Digits→ a sequence of digits

INTEGER CONSTANTS	
VALID	**INVALID**
2006 123 –902 +27 999 0 –6 +2	0.0 –15^{o}C 1,25,200 10 456 0.027 345+ 100.

THREE TYPES OF INTEGER CONSTANTS IN 'C'	
DECIMAL INTEGER CONSTANTS	• Any combination of digits from 0 to 9 • Preceded by optional + or – sign • Valid examples: 123, 109, –78, 0, 76594, +236
OCTAL INTEGER CONSTANTS	• Any sequence of one or more digits from 0 to 7 • An octal constant must start with 0 • It can be a positive or negative octal number with signs '+' or '–' • Valid examples: 027, –0126, +052, 0551
HEXA DECIMAL INTEGER CONSTANTS	• Any sequence of one or more digits from 0 to 9 inclusive of alphabets from 'A' to 'F' or 'a' to 'f'. • An octal constant must start with 0x or 0X • It can be a positive or negative hex number with signs '+' or '–' • Valid examples: 0X2, 0x9F, 0Xbcd, –0x23E

Floating Point Constants:

Floating point constants are real numbers with a decimal point embedded in it.

A floating point constant can be written in two forms:

1. Decimal or Fractional form:
   - It should contain atleast one digit to the right of the decimal point.
   - A '+' sign or a '–' sign may precede it. If the sign doesn't precede then the number is assumed to be positive.
   - The general form of the Decimal form of floating point constant is:

Sign	Integer part	Decimal point	Fractional part

where,

Sign	$\rightarrow$ optional plus or minus sign
Integer part	$\rightarrow$ a sequence of digits before the decimal point
Decimal point	$\rightarrow$ period symbol
Fractional part	$\rightarrow$ a sequence of digits after the decimal point

- Examples: 1.5 –0.563 123.0 27009.023 –999.99

2. <u>Exponential or Scientific form:</u>
   - The floating point constant in the exponential form can be expressed as:

Mantissa	e	Exponent

where,

Mantissa → is either a **real number** expressed in decimal form or an **integer**

e → alphabet ‘e’ (lower case) or ‘E’ (upper case)

Exponent → it is an integer number with an optional ‘+’ or ‘–’ sign

- Examples: 0.4876E12 –235.46e–102 +0.001E–97
  –43.2e06 156E+05 –1.234e–07

- Exponential notation is useful for representing numbers that are either very large or very small in magnitude. Example: 3500000000 may be written as 3.5E9 &–0.000000456 can be written as –4.56E–7

Character Constants:

- A character constant in ‘C’ is a single character enclosed within a pair of single quotes.
- Examples: ‘u’, ‘s’, ‘$’, ‘6’, ‘ ’, ‘?’, ‘!’, etc.
- Each character constant in ‘C’ is identified with its ASCII integer value.
- Character constants in ‘C’ are used mainly to represent some coding during data output.

String Constants:

- A string constant in ‘C’ denotes a sequence of one or more characters enclosed within a pair of double quotes.
- Examples: “Hi” “y2k problem” “15th August” “2006”
  “Welcome to C” “20% Loss”

- A single character string doesn’t have an equivalent integer value, but a single character constant has an integer value. Eg. “A” and ‘A’ are not the same.

Backlash Character Constants:

- 'C' supports some special backlash character constants that are used in output function printf to obtain special print effects.
- It is a combination of two characters in which the first character is always a backlash '\'. The second character can be any one of the characters a, b, f, n, t, v, ', ", \ and 0.
- They are also known as 'Escape Characters' or 'Escape Sequences'.
- A list of the Backlash Constants used are listed in table below:

Backlash Constants	**Meaning**
\a	System Alarm (Bell or Beep)
\b	Backspace
\f	Form feed
\n	New line (Line feed)
\r	Carriage return (CR)
\t	Horizontal tab (Fixed amount of space)
\v	Vertical tab
\"	Double quote
\'	Single quote
\?	Question mark
\0	Null character
\\	Backlash character itself

VARIABLES

A variable may be a data name that may be used to store data value.

- They are used to identify different program elements, hence also known as 'Identifiers'.
- A variable may take different values at different times during execution of the program.
- Each variable refers to specific memory locations in the Memory Unit where numerical values or characters can be stored.

- Example: sum = n1 + n2 here, n1 & n2 are variable names representing two different quantities in two distinct memory locations. Similarly, sum is also a variable name which holds the sum of n1 and n2.
- Other examples: area, amount, class_strength, condition, etc.

Rules for forming variable or identifier names:

1. The first character must be an alphabet (uppercase or lowercase) or an underscore "_"
2. All succeeding characters must be either letters or digits.
3. An ANSI standard recognizes a length of 31 characters. Many compilers treat only first 8 characters in the variable name as significant. (Presently, there is no limit on the variable name length).
4. Uppercase and lowercase identifiers are different in 'C' i.e., identifiers are case sensitive, so Sum SUM and sum each are different variable names.
5. No special characters or punctuation symbols are allowed except the underscore "_"
6. No two successive underscores are allowed, white spaces not allowed.
7. Keywords should not be used as identifiers.
8. Writing the variable names in lower case letters is a good practice.
9. Choose meaningful names to variables which reflect the functionality or nature of the variable.

Valid Variables	**Invalid Variables**
part_no , author x2 , ph_value total_marks_2006	2006salary , %sum (actual_bill) , ph value total__marks , long

DATA TYPES IN C

The Four fundamental data types:

'C' supports four basic data types:

Data type	**Keyword**	**Size (in bytes)**
Integer	int	2
Real (Floating point)	float	4

Double precision real	double	8
Character	char	1

int data type:

- int is the key word used to denote integer number.
- Integer constants in ‘C’ represent whole numbers without any fractional part.
- Integer numbers are stored on 16 bits (2 bytes).
- It is a sequence of digits without a decimal point.
- It can be prefixed with a plus or minus sign.
- When minus sign doesn’t precede the number, it is assumed to be a positive one.
- There should not be any special symbols like blank spaces, comma, etc.
- The range of integer number that can be stored depends on the word length of the computer. (word length: No. of bits accessed by processor at a time)
- For a 8 bit computer, range of int is given as:

  $-2^{8-1} \leq$ integer number $\leq +2^{8-1}-1$

  $-128 \leq$ integer number $\leq +127$

- For a 16 bit computer, range of int is given as:

  $-2^{16-1} \leq$ integer number $\leq +2^{16-1}-1$

  $-32786 \leq$ integer number $\leq +32767$

- Examples of valid int data type: 2006 123 –902 +27 999 0 –6 +2
- Examples of invalid int data type: 0.0 –15^{o}C 1,25,200 10 456 0.027 345+ 100.

float data type:

- float is the key word used to denote a real number or floating point number (both in fractional form and exponential form).
- Floating point numbers are stored in 32 bits (4 bytes) with 6 digits of precision.
- Examples: 1.5 –0.563 123.0 27009.023 –999.99 0.4876E12 –235.46e–102 +0.001E–97 –43.2e06 156E+05 –1.234e–07

double data type:

- double is the key word used to denote a double precision floating point number.
- This is similar to float data type, only difference is that it is used to represent more precision of the floating point number.
- Double precision floating point numbers are stored in 64 bits (8 bytes) with 16 digits of precision.

char data type:

- char is the keyword used to denote a character data type.
- A character constant in ‘C’ is a single character enclosed within a pair of single quotes.
- A character data type is stored on 8 bits (1 byte)
- Examples: ‘u’, ‘s’, ‘$’, ‘6’, ‘ ’, ‘?’, ‘!’, etc.
- Each character constant in ‘C’ is identified with its ASCII integer value.

Character	**ASCII**	**Character**	**ASCII**	**Character**	**ASCII**	**Character**	**ASCII**
‘A’	65	‘a’	97	‘0’	48	‘=’	61
‘Z’	90	‘z’	122	‘&’	38	‘{’	123

Data type modifiers:

The storage size in bytes and range of values being represented by basic data types can be modified with the help of the following **modifiers** or **qualifiers** as prefixes.

1. signed
2. unsigned
3. long
4. short

signed

- This modifier can be applied to integer variables although default integer declaration already assumes a signed number.

- In a signed data type, the first bit (known as the most significant bit) is used to store the sign. If this bit is 0, the number is positive. If it is 1, the number is negative.
- A signed int data type can hold values in the range –32786 to 32767
- When signed modifier is prefixed to char data type, it can store small integers in the range –128 to +127.

unsigned

- This modifier can be applied to both int and char data type variables.
- However, char data type is unsigned by default and can hold integer values in the range 0 to 255.
- An unsigned int data type can hold integer values in the range 0 to 65535

long

- This modifier can be applied to both int and double data types.
- When applied to int, it doubles the storage size from 2 bytes to 4 bytes.
- long int can store integer values in the range –2147483648 to +2147483647
- A long integer can also be declared as long int or simply long.
- When applied to double data type, the storage size increases from 8 bytes to 16 bytes.
- Long double can store floating point number values in the range –3.4E–4932 to +1.1E+4932

short

- This modifier can be applied to integer data type.
- It changes the size of int to its half. But since most of the compilers have 16 bits int, and the same size is maintained for short int also.

Modifier	**Size (Bytes)**	**Range of values**
int	2	–32768 to +32767
signed int	2	–32768 to +32767
unsigned int	2	0 to 65535
short int	2	–32768 to +32767
long int	4	–2147483648 to +2147483647
unsigned short int	2	0 to 65535
unsigned long int	4	0 to 4294967295

char	1	–128 to +127
signed char	1	–128 to +127
unsigned char	1	0 to 255
float	4	–3.4E+48 to +3.4E+48
double	8	–1.7E+308 to +1.7E+308
long double	16	–3.4E–4932 to +1.1E+4932

DECLARATION OF VARIABLES

- All the variables must be declared before they are used in a 'C' program
- Declarations are necessary to indicate to the compiler the type of variable and to reserve the amount of memory required to store the values held by them.
- The syntax for declaring a variable is as follows:

data type	variable list	semicolon

where,

data type	$\rightarrow$ Basic data type such as int, float, char, double, etc
variable list	$\rightarrow$ one or more variables of the above data type.
Semicolon	$\rightarrow$ a delimiter of this declaration.

Examples:

int count, i, j;

float average, sum, K[100];

char ch, ans;

double populn, cluster;

char city[30], train[20];

Assigning values to variables:

- The process of giving values to variables is known as "Assignment of values" to variables.

- The assignment operator ‘=’ is used to assign a value to a variable.
- The value assigned to the variable is stored in the memory location that was reserved for it during declaration.

Its syntax is:

Variable_name	=	value	;

Where,

Variable_name	$\rightarrow$ represents the name of the variable where it must be stored.
=	$\rightarrow$ is the assignment operator
value	$\rightarrow$ is a constant or a variable

Two methods of assigning values to variables:

1. Assigning initial values to variables within the declaration section. This is called **initialization**.

   Example: int i, gd = 10, gm;

   float n1, n2, sum = 0.0;char status, condn = ‘y’;

   char place[ ] = “NITTE”, college[50];

2. Assigning values to variables in the executable part of the program.

   Example: int k, num =30;

   float m;

   char ch;

   char name[50];

   Declaration

   k = num;

   m = 3.45;

   name = “MsBeautiful”;

   Assignment

ch = ‘N’;

Assignment Statements:

- An assignment statement has the following format (or syntax)

Variable_name	=	Expression	;

Where,

Variable_name	→ represents the name of the variable where it must be stored.
=	→ is the assignment operator
Expression	→ is

- constant or a variable
- function name or function call
- array reference or structure reference
- variables and/or constants coupled with operators

Examples of valid assignment statements:

```
bonous_pay = (basic_pay * 12)/100;

test_avg = (t1m + t2m + t3m)/3 ;

d = b * b – 4 * a * c;          y = num[8];

int_rate = 5.25;                big = largest(a,b,c);
```

Examples of invalid assignment statements:

```
x-y = x + y;

basic_pay + hra = gross_pay;

balance + 120 = int_rate * principal;

(a * a – b)/2.6 = d * d – c * c;

125 = reg_no;
```

Note: Expression must always come to the right side of the assignment operator '='

Short hand assignment operators:

'C' has special short hand operators which will simplifies representation of certain assignment statements.

Examples:

sum_of_digits_of_numbers=sum_of_digits_of_numbers+digit;

decrement_of_loop = decrement_of_loop – 1;

The above assignment statements can be written using **short hand operators** as:

sum_of_digits_of_numbers += digit;

decrement_of_loop –= 1;

Here the special operators '+=' and '-=' are short hand operators.

The general syntax for using short hand operators in assignment statements is:

Variable_name	operator =	expression	;

Short hand operator	**Meaning**
+=	Add and assign
–=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Find remainder and assign

Valid examples of short hand operators:

Expression with short hand operator	**Meaning**

sum += heights;	sum = sum + heights;
p += a-b;	p = p + (a-b);
down_count –= 1;	down_count = down_count – 1;
q -= 1+r;	q = q – (1+r);
x_to_pn *= a;	x_to_pn = x_to_pn * a;
A *= 2/(B*B);	A = A * 2/(B*B);
income /= tax;	income = income / tax;
xyz /= sd-mn*a;	xyz = xyz / (sd-mn*a);
profit %= (inv*4-loan);	profit= profit%(inv*4-loan);

Benefits of using short hand operators:

- Assignment expressions are concise and easy to use
- Considerable saving over length of source code
- The variable name used on LHS of '=' operator need not be re-typed in the RHS.

Multiple assignment statements:

'C' allows us to assign a single value to more than one variable name at a time.

Example: x = y = z = 0.0;

z is assigned a value 0.0, now value of z is assigned to y; and then value of y is assigned to x.

- In multiple assignment statements, the assignment is 'Right associative' as evaluation or assignment will take place from **right to left**.

Example 1: a = b = (x+y=z)/2;

1. evaluation of (x+y+z)/2 will take place first
2. then the result is assigned to variable name b
3. finally, value of b will be assigned to a

Example 2: l = m= n *= p;

l = (m= (n *= p)); ←(right to left)

1. n = n*p
2. m = n
3. l = m

Note: Multiple assignments cannot be done during declaration of variable names.

Therefore, int p = int q = int r = 0;

int x = y = z = 144;

$\Big\}$ ARE **INVALID** IN 'C'

(In the above statements first the variables must be declared and then multiple assignment statements may be used)

Symbolic Constants:

- 'C' allows defining a variable name as having a constant value using a preprocessor directive #define
- Such preprocessor statements are placed at the beginning of the program and are not a part of the C program.
- Such statements begin with the **# symbol** and hence do not end with a semicolon.
- A symbolic constant can be used to define a numeric constant or a character/string constant.
- Once defined, a symbolic constant's value can be used at many places in the program.

The syntax of symbolic constant is:

#define	symbolic_name	value of constant

Examples:

#definePI	3.141
#defineCLASS	"H Section"
#defineMINI_BAL	500.0
#defineFLAG	'Y'

Note:

- A symbolic constant should not be used as any other variable name

- Once a symbolic constant is defined, its value should can not be changed in the program. Example: MINI_BAL = 1000.0; is illegal.
- There should not be white space between # and define

Declaring variables as Constant and as Volatile: [ADDITIONAL INFORMATION]

const type modifier:

- It is sometimes required that the value of a variable should denote a single value through out the program execution.
- A qualifier or modifier 'const' is use to do so. This modifier 'const' must be placed before a data type declaration.

Examples:

```
const int max_count = 25;

const float int_rate = 6.25;

const char C = 'Y';
```

The above values of variables once initialized by the compiler cannot be changed.

volatile type modifier:

- It is sometimes required to dynamically change the value of a variable during program execution.
- The modifier or qualifier **volatile** is used for this purpose.

Examples:

```
volatile int score;

volatile float rate;
```

Delimiters:

- These indicate the boundary between the elements of a program
- They are also known as separators
- They separate constants, variables and statements.
- Comma, semicolon, single quotes, double quotes, blank spaces, etc. are the most commonly used delimiters.

Chapter 3: Operators and Expressions

- C supports a rich set of operators. Operators are used in programs to **manipulate data variables.**
- An operator is a **symbol** that tells the computer to perform certain mathematical or logical manipulation.
- Operators usually form a part of the **expressions** and indicate the type of expressions – **mathematical** or **logical expressions.**
- The values that can be operated by these operators are called **operands**.

Major categories of operators in 'C':

C OPERATORS			
Unary Operators	**Binary Operators**	**Ternary Operator**	**Special Operators**
• Unary Minus • Logical NOT • Bitwise Complementation	• Arithmetic Operators • Logical Operators • Relational Operators • Bitwise Operators	• A type of conditional operator in C	• Comma Operator • **sizeof()** operator • Address operator • Dot operator • Arrow operator

(A) Unary Operator:

An operator that acts on only one kind of operand is known as unary operator.
Types of unary operators are:
(1) Unary Minus (2) Logical NOT operator (3) Bitwise complementation

(1) Unary Minus:
Any positive operand with unary minus operator changes its value to negative. In effect, a positive number becomes negative, and a negative number becomes positive.
Example:

Let x = 10, y = 5
z = x + (-y)

= 10 + (-5) = 5

The logical NOT operator is considered later in Logical operators and bitwise complementation is considered in bitwise operators.

(B) Binary Operators:

These operators act on two operands and hence are named as binary operators. Following are the **four** types of binary operators:

(1) Arithmetic Operators	(3) Logical Operators
(2) Relational Operators	(4) Bitwise Operators

(1) ARITHMETIC OPERATORS

- The basic arithmetic operations addition, subtraction, multiplication and division can be performed on any built-in data type of C using these operators.
- Another operator, the modulus operator is added to the list of Arithmetic operators. It is used to find the remainder after integer division.

Operator	Meaning
+	Addition
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus or Modulo Division

Note:
(1) The modulus operator % can be used only for integer operands and not for floating point operands.
(2) C does not have an operator for exponentiation. This can be performed by a library function **pow()**.

Examples of arithmetic operators are:

a – b   a + b

where ‘a’ and ‘b’ are operands

```
a * b        a / b
a % b        -a * b
```

(a) Integer Arithmetic:

- When both the operands in a single arithmetic expression are integers, the expression is called **integer expression** and the operation is called **integer arithmetic**.
- Integer arithmetic always yields an integer value.

Examples: if a = 17 and b = 4, the following are the results of integer arithmetic.

```
a – b = 13
a + b = 21
a * b = 68
a / b =  4 (decimal part truncated)
a % b =  1 (remainder of division)
```

```
/**** Use of Modulus Operator ****/
#include <stdio.h>

int main()
 {
        int months,tot_days,days;
        printf("\n\n\tEnter the no. of days:");
        scanf("%d",&tot_days);
        months=tot_days/30;
        days=tot_days%30;
        printf("\n\n\t%d days = %d months & %d days", tot_days, months, days);
        return 0;
 }
```

Output of Program:

```
Enter the no. of days: 200


200 days = 6 months & 20 days
```

(b) Floating Point or Real Arithmetic:

- An arithmetic expression involving only real (or floating point) operands is called Real Arithmetic.
- A real operand may assume values either in the decimal or exponential form.
- The result of real arithmetic operation is an approximation of the correct result to the number of significant digits permissible.

- The modulus operator % **cannot be** used with real operands.

Examples: if x = 8.3, y = 3.2 and z are floats, then we will have:

z = x + y = 11.500000
z = x / y = 2.593750
z = -y/x = -0.385542

(c) Mixed-mode Arithmetic:

- The arithmetic operation in which one of the operand is real and the other is integer, is called mixed-mode arithmetic operation.
- The value of such an expression is a float.
- But if the result is assigned to an int variable, the decimal portion of the result is truncated and only integer portion is assigned to the int variable.
- Example: int n;

  n = 1 + (1.5) = 2.5

  But 'n' is an integer variable, so only the integer part of 2.5 i.e., 2 is assigned to 'n'. Therefore, n = 2.

Arithmetic Expressions:

- An expression involving arithmetic operators is called as an arithmetic expression.
- These expressions connect one or more operands (integer or real) through arithmetic operators.
- The conventional mathematical expressions that we normally write must be converted into equivalent C expressions as C compiler understands only the symbols provided in the C character set.

Some sample expressions and their C equivalent expressions are:

Mathematical Expression	**C Equivalent**
$\frac{a+b}{a-b}$	(a+b)/(a-b)
$\frac{2x^2}{p+q} \times (1-m^2)$	(2*x*x/(p+q))*(1-m*m) OR (2*x*x*(1-m*m))/(p+q)

$T = \frac{2m_1m_2}{m_1+m_2} \times g$	T = 2*m1*m2*g/(m1+m2) OR
	T = (2*m1*m2/(m1+m2))*g

$side = \sqrt{a^2 + b^2 - 2ab\cos(x)}$	side=sqrt(a*a+b*b-2*a*b*cos(x))
$E = m\left[gh + \frac{v^2}{2}\right]$	E=m*(g*h+v*v/2)
$e^{\lvert a \rvert} + b$	exp(abs(a))+b // **abs()** returns the **absolute value** of an integer.
$x = e^{\left\lvert \frac{y}{\sqrt{1+\sin\theta}} \right\rvert}$	X=exp(abs(y/sqrt(1.0+sin(theta))))

Evaluation of Arithmetic expressions:

- Arithmetic expressions are evaluated from left to right
- Operands associated with highest priority are operated first

Arithmetic Operations	**Priority**
Multiplication, Division and Modulus	Highest priority
Addition and Subtraction	Lowest priority

Rules for evaluation of arithmetic expressions:

- If the given expression involves parentheses, then the expression inside the parentheses must be evaluated first.
- The parenthesized and unparenthesized expressions follow the operator precedence as given in table above.
- If a unary minus is present in the expression, then the term associated with unary minus must be evaluated before any other expressions.

Example:

```
2*((i/3)+4*(j-2))          // given i=8 and j=5;
=2*((8/3)+4*(5-2))
=2*(2+4*(5-2))
=2*(2+4*3)
=2*(2+12)
=2*14
=28
```

INCREMENT AND DECREMENT OPERATORS

- C has two very useful operators generally not found in other languages.
- They are: **Increment Operator** denoted by **++**
  Decrement Operator denoted by **--**
- The increment operator ++ adds 1 to the operand while -- subtracts 1 from the operand associated with it.
- These operators act only on integer operands.
- Both are **unary operators** taking the form

++m or **m++**	which is equivalent to	**m = m + 1**
--m or **m--**	which is equivalent to	**m = m - 1**

- Increment and Decrement Operators are used extensively in **for** and **while** loops.
- ++m and m++ (or --m and m--) mean the same when they are used independently in statements which are not expressions.
- They behave **differently** when they are used in **expressions** on right-hand side of an assignment statement as shown in table below.

Expression	Meaning	Known as

x = ++a;	**a = a+1;**
	x = a;
		Pre increment – Integer variable on RHS will be incremented first, then it's new value is assigned to variable on LHS.

x = a++;	**x = a;**
	a = a+1;
		Post increment – Present value of integer variable on RHS is first assigned to variable on LHS, then integer on RHS is incremented.

y = --b;	**b = b-1;**
	y = b;
		Pre decrement– Integer variable on RHS will be decremented first, then it's new value is assigned to variable on LHS.

y = b--;	**y = b;**
	b = b-1;
		Post decrement– Present value of integer variable on RHS is first assigned to variable on LHS, then integer on RHS is decremented.

```
/*** Use of increment and decrement operators ***/
#include <stdio.h>

int main()
{
        int a=10,b=12;
        printf("Given a = %d and b = %d",a,b);
        printf("\na++ = %d",a++);
        printf("\nb-- = %d",b--);
        printf("\nNow a = %d",a);
        printf("\nNow b = %d",b);
        printf("\n++a = %d",++a);
        printf("\n--b = %d",--b);
        return 0;
}
```

Output of the program:

```
Given a = 10 and b = 12

a++ = 10

b-- = 12

Now a = 11

Now b = 11

++a = 12

--b = 10
```

(2) RELATIONAL OPERATORS

- These are used to compare two operands. For example, compare the ages of two persons, compare the prices of two or more items, etc.
- They result in either a TRUE (Non-zero) value or FALSE (Zero) value.

Operator	Meaning	Precedence	Associativity
<	Lesser than	1	L to R
<=	Less than or equal to	1	L to R
>	Greater than	1	L to R
>=	Greater than or equal to	1	L to R
= =	Equal to	2	L to R
!=	Not equal to	2	L to R

Examples: **x > y**
age = = 25
cost <= 125.5

i != 10
days < 365

Relational Expressions:

- A simple relational expression contains only one relational operator. Its syntax is a follows:

ae1	relational_operator	ae2

where, ae1 and ae2 are arithmetic expressions, which may be simple constants, variables or combination of them.

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.
- Relational expressions are used in ***decision statements*** such as **if** and **while** to decide the course of action of a running program.

Suppose a and b are two float point variables holding values 4.5 and 10.0,

Expression	Interpretation	Value
a <= b	TRUE	1
a < -b	FALSE	0
a*9 > 0	FALSE	0
a + 5 < b * 2	TRUE	1
a*b == b*a	TRUE	1

```
/*** Program to illustrate the use of relational operator ***/
#include <stdio.h>

int main()
{
        int a, b;
        float n1, n2;
        printf("\n\nEnter two integers and two real numbers:");
        scanf("%d %d %f %f",&a, &b, &n1, &n2);

        if(a = = b)
```

```
            printf("\n\n a and b are equal");
        else
            printf("\n\n a and b are unequal");

        if(n1 > n2)
            printf("\n\n n1 is greater than n2");
        else
            printf("\n\n n1 is not greater than n2");

        return 0;
}
```

Output of Program:

```
1. Enter two integers and two real numbers: 10      20      2.5     0.25
    a and b are unequal

    n1 is not greater than n2

2. Enter two integers and two real numbers: 6       6       25      12.5
    a and b are equal

    n1 is greater than n2
```

(3) LOGICAL OPERATORS

- Logical operators are used in C for decision making.
- C has the following **three** logical operators:

Operator	Meaning	Precedence	Associativity
&&	Logical AND	2	L to R
||	Logical OR	3	L to R
!	Logical NOT	1	L to R

- && and || are binary operators and ! is a unary operator.
- The result of these operators is either TRUE (ONE) or FALSE (ZERO).

- The logical operators are used to connect one or more relational expressions.
- Such an expression which combines two or more relational expressions is called as **logical expression** or **compound relational expression**.
- While using logical AND,
  - If both the operands are TRUE, then the result of the logical expression is TRUE.
  - Even if one of the operands is FALSE, then the result of the logical expression is FALSE.
- While using logical OR
  - Only if both the operands are FALSE, then the result of the logical expression is FALSE.
  - Even if, one of the operands is TRUE, then the result of the logical expression is TRUE.
- While using logical NOT, the compliment of the operand is obtained. That is, if the operand is TRUE, than the result will be FALSE or vice-versa.
- The results of the operands are shown in Truth table shown below:

Op1	**Op2**	**Value of Expression**			
		Op1 && Op2	**Op1 || Op2**	**!Op1**	**!Op2**
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Examples of Logical Expressions:

1. **if(age > 55 && salary < 50000)**
2. **if(number < 0 || number > 100)**

Suppose a, b and c are integers having values, 2, 4 and 3,

a && b || c && (!b)
= 2 && 4 || 3 && (!4)
= 2 && 4 || 3 && 0
= 1|| 3 && 0
= 1||0
= 1

/*** PROGRAM TO ILLUSTRATE USE OF LOGICAL OPERATORS ***/

```
#include <stdio.h>

int main()
{
        int a,b,c;
        a = 10; b = 5;
        c = a && b;
        b = a || b || c;
        a = a && b || c;
        printf("\n\n%d %d %d",a,b,c);
        return 0;
}
```

Output of the program:

```
1 1 1
```

(4) BITWISE OPERATORS IN ‘C’

- All the data stored in the computer memory are in sequences of bits (0’s and 1’s).
- Some applications require the manipulation of these bits.
- Manipulation of individual bits is carried out in machine language or assembly language.
- ‘C’ provides **six operators** to perform **bitwise operations**.
- These operators work only with **int** and **char** data-types. They cannot be used with floating point numbers.

The **six** bitwise operators in ‘C’ are:

Operator	**Meaning**
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR (XOR)
~	1’s complement
<<	Left shifting of bits
>>	Right shifting of bits

(a) Bitwise AND:

b1	**b2**	**b1 & b2**

Result of bitwise AND is 1 when both the bits are 1, otherwise

0	0	0
0	1	0
1	0	0
1	1	1

Examples:

1. Considering normal binary digits.

```
int a = 4, b = 3;
Equivalent binary value of a = 4 is        0000 0100
Equivalent binary value of b = 3 is        0000 0011
-------------------------------------------------------
        a & b                 is           0000 0000
-------------------------------------------------------
```

2. Considering BCD (Binary Coded Decimal).

```
int m = 120, n = 060, p;
Equivalent BCD of m = 120 is    001 010 000
Equivalent BCD of n = 060 is    000 110 000
-------------------------------------------------------
        p = m & n             is    000 010 000
-------------------------------------------------------
```

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

(b) Bitwise OR (|):

Result of bitwise OR is 1 when one of the bits is 1, otherwise (when both bits are 0s) it is zero.

b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

Examples:

1. Considering normal binary digits.

```
int a = 6, b = 4;
Equivalent binary value of a = 6 is        0000 0110
Equivalent binary value of b = 4 is        0000 0100
-------------------------------------------------------
        a | b                 is           0000 0110
-------------------------------------------------------
```

2. Considering BCD (Binary Coded Decimal).

```
int m = 340, n = 723, p;
```

Equivalent BCD of m = 340 is		011 100 000
Equivalent BCD of n = 723 is		111 010 011
p = m | n	is	111 110 011

(c) Exclusive OR (XOR ^) :

Result of bitwise XOR is 1 if the bits are different (1 and 0 or 0 and 1), otherwise (when both bits are 0's and both bits are 1's) it is zero.

b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

Examples

Considering normal binary digits.

int a = 6, b = 4;

Equivalent binary value of a = 6 is		0000 0110
Equivalent binary value of b = 4 is		0000 0100
a ^ b	is	0000 0010

(d) One's Complement (~):

- The bitwise complement operator is an unary operator which reverses the state of each bit within an integer or character.
- Each 'zero' get changed to 'one' and each 'one' gets changed to 'zero'.

Example:

int b = 12;

= 1100 (in binary form)

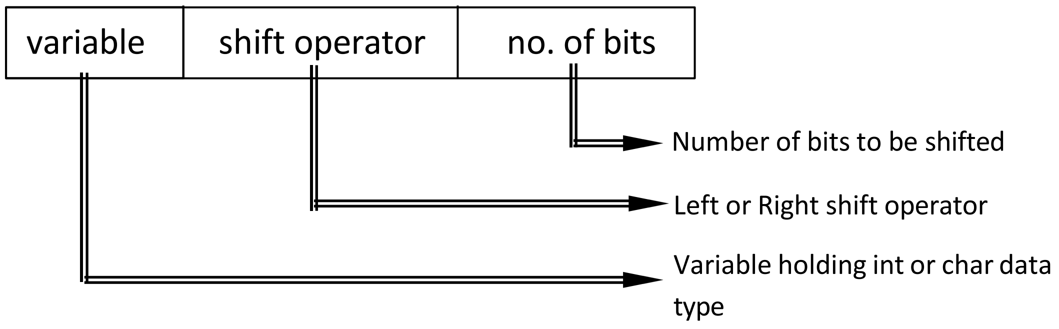
∴ a = ~b

= 0011

(e) Left Shift Operator (<<) and Right Shift Operator (>>):

The left shift operator << shifts the bits to the left and the right shift operator >> shifts the bits to the right.

General syntax for bitwise shift operator:


variable
shift operator
no. of bits
Number of bits to be shifted
Left or Right shift operator
Variable holding int or char data type

Examples: m >> 2
n << 1

Example for bitwise left shifting:

Consider an integer variable a = 32

In binary form of 32 =

0	0	1	0	0	0	0	0

a << 2 = (128) =

1	0	0	0	0	0	0	0

Example for bitwise right shifting:

Consider an integer variable a = 32

In binary form it is a =	0	0	1	0	0	0	0	0
a >> 2 = (8)	0	0	0	0	1	0	0	0

```
/*** PROGRAM TO ILLUSTRATE USE OF BITWISE SHIFT OPEATORS ***/
#include <stdio.h>

int main()
  {
        unsigned int x,y;
        x = 128; y = 32;
        printf("\n\nBefore right shifting, x = %d",x);
        printf("\nBefore left shifting, y = %d",y);
        x = x >> 2;
        y = y << 3;
        printf("\n\nAfter right shifting x by 2, x = %d",x);
        printf("\nAfter left shifting y by 3, y = %d",y);
        return 0;
}
```

Output of the program:

```
Before right shifting, x = 128

Before left shifting, y = 32

After right shifting x by 2, x = 32

After left shifting y by 3, y = 256
```

(C) TERNARY/CONDITIONAL OPERATOR

- It takes three operands.
- There is only one such operator in 'C' and it is called the '**Conditional Operator**'.
- It uses character pair **? :**
- Its general syntax is

test_exp ? exp1 : exp2

where, **test_exp** is a test condition, usually a relational expression like a > b
exp1 and **exp2** can be any valid arithmetic expressions or variables or constants.

- The **? :** operator pair works as follows:
    - If the result of the test_exp is TRUE, exp1 is evaluated and the value of exp1 is the value of conditional operation.
    - If the value of test_exp is FALSE, exp2 is evaluated and the value of exp2 is the value of the conditional operation.

Examples:

(1) m = x > y ? a : b - 146;

if x > y is TRUE, the value of '**a**' is assigned to '**m**',
or if x > y is FALSE, the value of '**b-146**' is assigned to '**m**'.

(2) flag = (c == 'y' || c == 'Y') ? 1 : 0;

if (c == 'y' || c == 'Y') is TRUE, **1** is assigned to '**flag**'
or if (c == 'y' || c == 'Y') is FALSE, **0** is assigned to '**flag**'.

```
/*** THE USE OF CONDITIONAL OPERATOR ***/
#include <stdio.h>

int main()
 {
      int a,b,min,max;
      printf("\n\nEnter two integer numbers:");
      scanf("%d %d",&a,&b);
      printf("\n\na = %d        b = %d",a,b);
      max = a > b ? a : b;
      min = a < b ? a : b;
      printf("\n\nMaximum of a and b is %d",max);
      printf("\nMinimum of a and b is %d",min);
      return 0;
 }
```

Output of the program:

```
Enter two integer numbers: 23 56


a = 23     b = 56


Maximum of a and b is 56

Minimum of a and b is 23
```

(D) SPECIAL OPERATORS

(i) The sizeof() Operator:

- The **sizeof()** operator returns the size (number of bytes) of the operand.
- The operand may be a constant, variable or any valid data-type.
- The operand is written within the parentheses of the sizeof operator.

- It is commonly used to determine the lengths of arrays and structures when their sizes are not known to the programmer.
- It is also used to allocate memory space dynamically to variables during the execution of the program.

Examples:

```
a = sizeof (int);
y = sizeof (avg);
```

```
/**** USE OF SIZEOF() OPERATOR ****/

#include <stdio.h>

int main()
{
        int x;
        float y;
        char ch = 'y';
        x = 10; y = 100.0;
        printf("\n\nSize of x = %d",sizeof(x));
        printf("\nSize of y = %d",sizeof(y));
        printf("\nSize of ch = %d",sizeof(ch));
        printf("\nSize of double = %d",sizeof(double));
        return 0;
  }
```

Output of the program:

```
Size of x = 2

Size of y = 4

Size of ch = 1

Size of double = 8
```

(2) The Comma Operator:

- The comma operator can be used to link the related expressions together.
- A comma linked list of expressions are <u>evaluated left to right</u> and the value of the <u>right most expression</u> is the value of the combined expression.

Examples:

(1) value = (x = 10, y = 5, x+y);	first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e., 10+5) to **value**.
(2) temp = a, a = b, b = temp;	the comma operator is also used to exchange the values stored in two memory locations.

(3) if(n1 > n2) temp = n1, n1 = n2, n2 = temp;

- It is also used in for loops
  to <u>initialize</u> two variables simultaneously and
  also <u>increment or decrement</u> two variables simultaneously.

Examples:
(1)for(n = 1, m = 10; n <= m; n++, m++)
(2)for(i = 0; i < 10; k--, ++i)

- The comma operator can also be used in while loop statements.

Examples:
(1) while(r = pnum % z, r != 0)
(2) while(ch = getchar(), ch == ‘y’)

```
/**** USE OF COMMA OPERATOR ****/

#include <stdio.h>
int main()
 {
      int a,b,temp;
      b = (a = 20,a + 10);
      printf(“\n\nBefore swapping, a = %d & b = %d”,a,b);
      temp = a, a = b, b = temp;
      printf(“\nAfter swapping, a = %d & b = %d”,a,b);
      return 0;
 }
```

Output of the program:

```
Before swapping, a = 20 & b = 30

After swapping, a = 30 & b = 20
```

Shorthand Assignment Operators:

- Shorthand assignment operators associated with arithmetic operations have already been seen.
- The bitwise operators can also be written in shorthand form.

The shorthand arithmetic operators and bitwise operators:

Operator	Assignment Expression	Shorthand Assignment
+ (plus)	a = a + b	a += b
- (minus)	a = a – b	a -= b
* (asterisk)	a = a * b	a *= b
/ (slash)	a = a / b	a /= b
% (percentage)	a = a % b	a %= b
& (bitwise AND)	a = a & b	a &= b
| (bitwise OR)	a = a | b	a |= b
^ (XOR)	a = a ^ b	a ^= b
<< (left shift)	a = a << b	a <<= b
>> (right shift)	a = a >> b	a >>= b

C EXPRESSIONS

Expressions are basically operators acting on operands. An expression in C is defined as 2 or more operands are connected by one operator. All operators discussed above can form different expressions.

When an expression can be interpreted in more than one way, there are rules that govern how the expression gets interpreted by the compiler. Such expressions follow C's precedence and associativity rules. The precedence of operators determine a rank for the operators.

PRECEDENCE AND ASSOCIATIVITY OF C OPERATORS

Operator	Description	Associativity	Precedence
() **[]**	Function call Array references	L to R	1
+ **-** **++** **--** **!** **~** *****	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference	R to L	2

& **sizeof()** **(type)**	Address Size of an object Type cast (conversion)		
***** / **%**	Multiplication Division Modulus	L to R	3
+ **-**	Addition Subtraction	L to R	4
<< >>	Bitwise left shift Bitwise right shift	L to R	5
< <= > >=	Less than Less than or equal to Greater than Grater than or equal to	L to R	6
== **!=**	Equal to Not equal to	L to R	7
&	Bitwise AND	L to R	8
^	Bitwise XOR	L to R	9
|	Bitwise OR	L to R	10
&&	Logical AND	L to R	11
||	Logical OR	L to R	12
? :	Conditional expression	R to L	13
= ***=** /= **%=** += **-=** **&=** ^= |= <<= >>=	Assignment operators	R to L	14
,	Comma operator	L to R	15

For example, the expression a * b + c can be interpreted as (a * b) + c or a * (b + c), but the first interpretation is the one that is used because the multiplication operator has higher precedence than addition.

TYPE CONVERSION IN C

- The process of **converting one type of data type to another**, say change **int** to **float** or **float** to **int** is known as **data type conversion** or **type casting**.
- The general form of type casting is

  Syntax: **(data type)variable**

Example:

```
int a = 3;
float sum;
sum = 20/a;
………………………………
```

Value of 'sum' is 6.00000

```
int a = 3;

float sum;

sum = 20/(float)a;
```

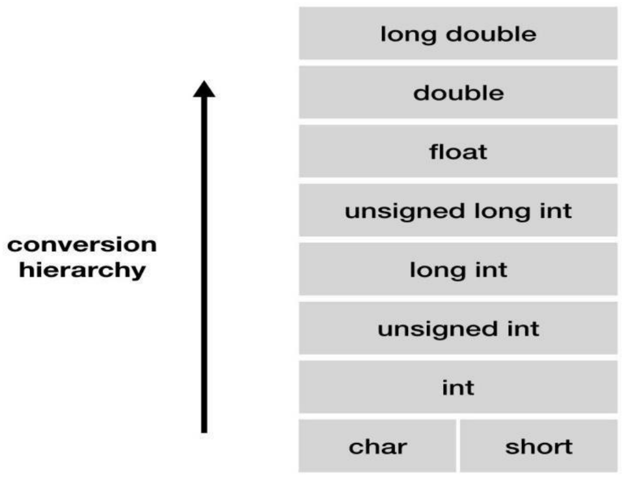
Value of 'sum' is 6.666666

Data type conversion can be done in two ways:

1. Implicit type conversion
2. Explicit type conversion

Implicit type conversion:

This type of conversion is usually performed by the compiler when necessary without any commands by the user. Thus it is also called "Automatic Type Conversion". The compiler usually performs this type of conversion when a particular expression contains more than one data type. The compiler first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy.



Example:

```
#include <stdio.h>

int main()
{
	int  i = 17;
	char c = 'c'; /* ascii value is 99 */
	float sum;
	sum = i + c;
	printf("Value of sum : %f\n", sum );
}
```

Output:

```
Value of sum : 116.000000
```

Here, first c gets converted to integer, but as the final value is double, usual arithmetic conversion takes place and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

Explicit type conversion:

In Explicit type conversion, the user explicitly defines within the program the datatype, to which the operands/variables of the expression need to be converted.

Example:

```
#include <stdio.h>

int main()
{
        double a = 4.5;
        double b = 4.6;
        double c = 4.9;
        int result;
        result = (int)a + (int)b + (int)c; //explicitly defined by user
        printf("Result = %d", result);
        return 0;
}
```

Output:

```
Result = 12
```

Here, output result is 12 because in the resulting expression the user has explicitly defined the operands (variables) as integer data type. Hence, there is no implicit conversion of data type by the compiler.

(Additional Information)

Mathematical functions:

- C provides a large number of mathematical functions which readily calculate the values of trigonometric, hyperbolic and other functions.
- All these functions are defined in the header file **math.h**.
- Some of the most commonly used mathematical functions are:

Function		**Return type**	**Meaning**
Trigonometric	acos(x)	double	Arc cosine of x ($\cos^{-1}$ x)
	asin(x)	double	Arc sine of x ($\sin^{-1}$ x
	atan(x)	double	Arc tangent of x ($\tan^{-1}$x)
	atan2(x,y)	double	Arc tangent of (x/y) [$\tan^{-1}$(x/y)]
	cos(x)	double	Cosine of (x)
	sin(x)	double	Sine of (x)

	tan(x)	double	Tangent of (x)

Hyperbolic	cosh(x)
	sinh(x)
	tanh(x)
		double
		double
		double
			Hyperbolic cosine of x
			Hyperbolic sine of x
			Hyperbolic tangent of x

Other functions	abs(x)
	ceil(x)
	exp(x)
	fabs(x)
	floor(x)
	fmod(x,y)
	log(x)
	log10(x)
	pow(x,y)
	sqrt(x)
		int
		double
		double
		double
		double
		double
		double
		double
		double
		double
			Absolute value of x
			x rounded up to nearest integer
			e to the power of x (e^x)
			Absolute value of x
			x rounded down to nearest integer
			Remainder of x/y
			Natural log of x, x > 0 ($\log_e x$)
			Base 10 log of x, x > 0 ($\log_{10} x$)
			x to the power of y (x^y)
			Square root of x, x >= 0

```
/*** USE OF MATHEMATICAL FUNCTIONS ***/
#include  <stdio.h>
#include  <math.h>
#include <conio.h>
#define PI 3.141

int main()
 {
        float x = 40, y = 30; clrscr();
        printf("\n\nx = %f  &  y = %f",x,y);
        printf("\n\nSquare Root of x = %f",sqrt(x));
        printf("\nSine of x = %f",sin(x*PI/180));
        printf("\nCosine of y = %f",cos(y*PI/180));
        printf("\nTangent of y = %f",tan(y*PI/180));
        printf("\n8th power of x = %f",pow(x,8));
        printf("\nLogarithm of x to base 10 = %f",log10(x));
        return 0;
       }
```

Output of the program:

```
x = 40.000000    &     y = 30.000000

Square Root of x = 6.324555

Sine of x = 0.642687

Cosine of y = 0.866075

Tangent of y = 0.577219

8th power of x = 6553600000000.000000

Logarithm of x to base 10 = 1.602060
```

Chapter 4: Managing Input and Output Operations

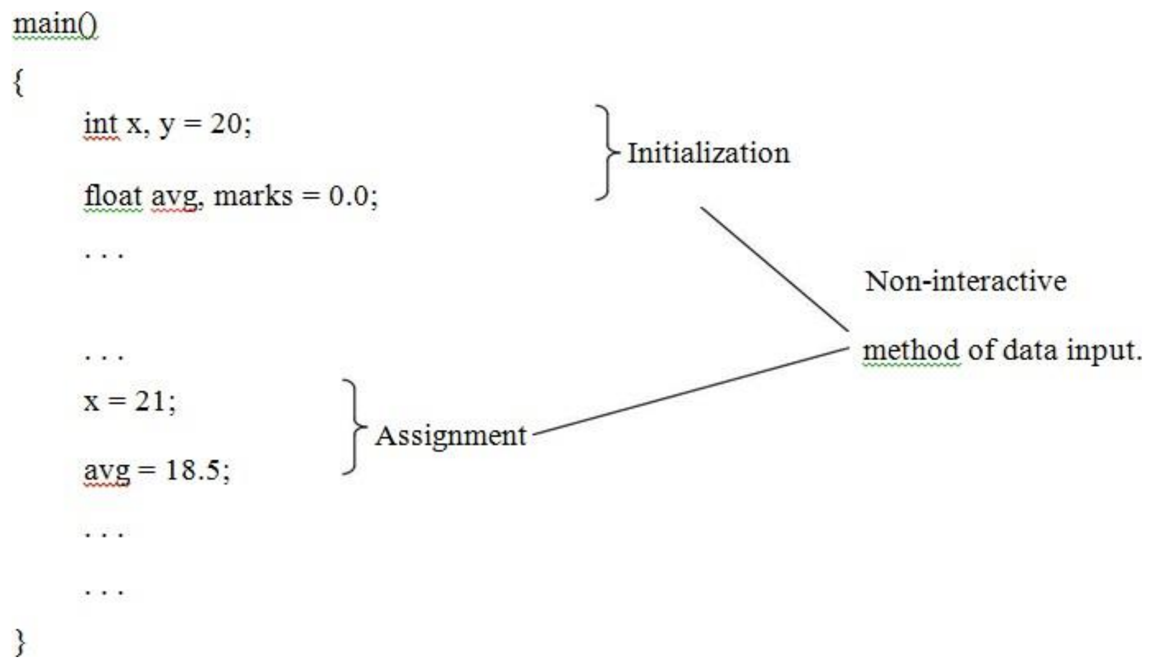
- Reading, processing and writing are the three essential functions of a computer.
- Every computer program has to take some data as **input** and write/print/display the processed data as **output**.

There are two methods of data input:

1. Non-Interactive method:

Values are assigned to the variables in the program itself.

Example:



```
main()
{
        int x, y = 20;
        float avg, marks = 0.0;
        . . .

        . . .
        x = 21;
        avg = 18.5;
        . . .
        . . .
}
```

2. Interactive Method:

Data is supplied to the computer by the user through standard input device like the key board.

Examples are the input functions like **scanf**, **getchar**, **gets**, etc. of ‘C’ which are used to read data into the computer.

INPUT – OUTPUT FUNCTIONS

- ‘C’ is a functional programming language. i.e., a ‘C’ program is constructed by making use of functions. ‘C’ provides a set of functions to perform the basic input/output (I/O) operations.
- All these I/O functions are stored in a header file by name **standard input output library** and is denoted by **stdio.h**
- We attach this header file to our C program by using the preprocessor directive **#include <stdio.h>** at the beginning of the program.

  #include <stdio.h> tells the compiler to search for the file named **stdio.h** and place its contents at this point in the program. The contents of the header file become a part of the source code when it is compiled.
- Some of the standard I/O functions frequently used in ‘C’ are:

scanf()	printf()	putchar()
getchar()	getch()	getche()
gets()	puts()	

- Note that the functions getch() and getche() are defined in another header file by name **conio.h**

There are two types of I/O functions in ‘C’:

Formatted I/O functions: These enable the user to specify the type of data and the way in which it should be read or written out.

Unformatted I/O functions: Using these types of functions, the user cannot specify the type of data and the way in which it should be read or written out.

	Input function/s	**Output function/s**
Formatted	scanf()	printf()
Unformatted	getchar(),getch(), gets(), getche(), etc.	putchar(), puts() etc.

FORMATTED INPUT

It refers to the input of one or more data types arranged in a particular format.

For example, consider

23.06 192 Nitte

A single line consists of three pieces of data arranged in a particular format.

The first part 23.06 must be read into a float variable.
The second part 192 must be read into an integer variable
The third part "Nitte" must be read into a string variable.

Such data must be read according to the sequence of their appearance. This can be done using the input function **scanf()** in 'C'.

For the above example the **scanf()** function is written as:

```
scanf("%f %d %s",&weight,&height,name);
```

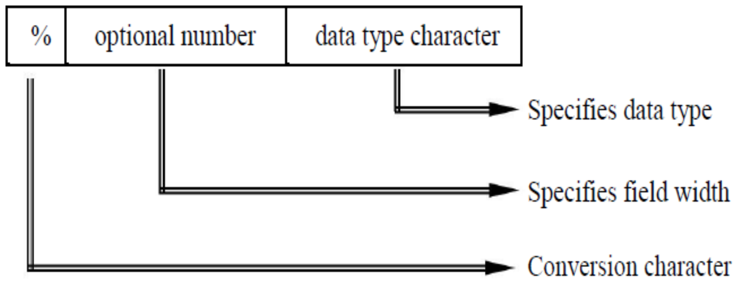
Syntax of scanf() function:

```
scanf("control string", address_list);
                    OR
scanf("control string", &var1, &var2,………..&varn);
```

"control string" contains one or more character groups (or field specifiers) which direct the interpretation of input data.

address_list OR **&var1, &var2,…** ....**&varn** specify the address locations of the variables where the data is stored.

Each character group (field specifier) starts with a % symbol.



Without the optional number which specifies the field width, the character group format is

%	data type character

The table below shows a list of the character groups (or field specifiers) commonly used:

Character group	Meaning
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating point number

%f	Read a floating point number
%g	Read a floating point number
%h	Read a short integer
%i	Read a decimal or hexadecimal or octal integer
%o	Read an octal number
%p	Read a pointer
%s	Read a string
%u	Read an unsigned integer
%x	Read a hexadecimal number

Examples of scanf()functions:

```
scanf("%d%f%c", &n1,&n2,&ch);
scanf("%d,%f,%c",&num,&avg,&c1);
scanf("%d  %f       %s",&a,           &total, msg);
```

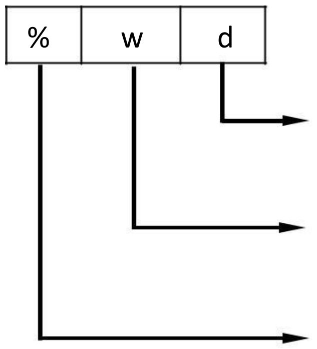
Rules that govern the syntax of scanf():

- Control string must be enclosed with double quotes.
- For every input variable there must be one character group.
- Multiple number of character groups are allowed in a control string. Theymust be separated by blank spaces.
- Each input variable must in the address_list must be preceded by an ampersand (&) symbol.
- White spaces may be included in the address_list, but all variables in the address list must be separated by commas.

- Address_list must not be enclosed within double quotes.
- The number of character groups, data type and sequence in the control string should match those of the variables in the address_list.
- A comma should separate the control string and the address_list.

Reading Integer numbers:

The character group (or the field specifier) for reading the integer number is:



Data type character (d) – indicates that the number to be read is in decimal integer mode.

An integer number (w) – it specifies the field width of the number to be read.

The conversion character (%) – it indicates that the conversion specification follows.

Consider the following example:

scanf("%2d %5d",&num1,&num2);

When the computer executes this statement, it waits for the user to enter values for the variables num1and num2.

Let the values entered be: 50 31426.
50 will be assigned to num1and 31426 to num2.

Suppose the values entered are: 31426 50

num1 will be assigned with 31 (because of the field width %2d)

num2 will be assigned with 426 (unread part of 31426)

The value 50 that is unread will be assigned to the first variable in the next **scanf()** statement.

These kinds of errors can be eliminated if we use <u>character groups</u> without specifying the field width (w).

Thus the statement **scanf("%d %d",&num1,&num2);**

will read the data 3142650 correctly and assign 31426 to num1and 50 to num2.

If we enter a floating point number instead of an integer, the fractional part may be ignored and **scanf()** may skip reading further input.

In place of the optional field width specifier (w) if we use a **asterisk * ** symbol, that particular input field will be skipped.

For example, **for the input statement scanf("%d %*d %d",&a,&b,&c);**

if we enter 123 45 97, 123 will be assigned to a and 97 will be assigned to b, <u>45 will not be assigned to b</u> and will be skipped. garbage value will be assigned to c.

The data type character '**d**' may be preceded by the letter '***l***' (i.e., % ***l*d**) to read long integers.

Example:

```
/******READING INTEGER NUMBERS**************/

#include <stdio.h>

int main()

{
     int a,b,c,x,y,z;
```

```
        int p,q,r;

        printf(“\n\n\tEnter three integer numbers:”);
        scanf(“%d %*d %d”,&a,&b,&c);
        printf(“\n\t                a=%d          b=%d    c=%d”,a,b,c);

        printf(“\n\n\tEnter two 4-digit integer numbers:”);
        scanf(“%2d %4d”,&x,&y);
        printf(“\n\t                x=%d          y=%d”,x,y);


        printf(“\n\n\tEnter two integer numbers:”);
        scanf(“%d %d”,&a,&x);
        printf(“\n\t                a=%d x=%d”,a,x);

        printf(“\n\n\tEnter a nine digit integer number:”);
        scanf(“%3d %4d %3d”,&p,&q,&r);
        printf(“\n\t                p=%d          q=%d          r=%d”,p,q,r);


        printf(“\n\n\tEnter two 3-digit integer numbers:”);
        scanf(“%d %d”,&x,&y);
        printf(“\n\t                x=%d          y=%d”,x,y);

     return 0;

   }
```

Output of program:

```
Enter three integer numbers:

1        2        3
a=1      b=3      c=-29730


Enter two 4-digit integer numbers:

6789            4321

x=67            y=89


Enter two integer numbers:

44      66
a=4321 x=44

Enter a nine digit integer number:

123456789

p=66            q=1234                  r=567


Enter two 3-digit integer numbers:

123             456

 x=89 y=123
```

Reading real (floating point) numbers:

- Real numbers (both fractional and exponential notations) can be input by using

  either of the character groups %f %e %g

  For example, scanf("%f %f %f",&x,&y,&z);

  with the input data 475.89 43.21E-1 678

  will assign 475.89 to x, 4.321to y and 678.0to z

- If the number to be read is a double precision one, then the character group ***%lf*** instead of ***%f*** must be used.

- A number may be skipped by using the **%*f** specification.

```
/********READING OF REAL NUMBERS**************/
#include <stdio.h>
#include <conio.h>

int main()
{
        float x,y;
        double p,q;
        printf("\n\n\tENTER THE VALUES OF X AND Y:");
        scanf("%f %g",&x, &y);
        printf("\n\t x=%f\n\ty=%f",x,y);
        getch();

        printf("\n\n\t                    ENTER THE VALUES OF P AND Q:");
        scanf("%lf %lf",&p,&q);
        printf("\n\t p=%.12lf\n\tq=%.10e",p,q);
        return 0;
}
```

Output of program:

```
ENTER THE VALUES OF X AND Y:12.3456                              12.3e-2
x=12.345600
```

```
y=0.123000

ENTER THE VALUES OF P AND Q:1.123456789                    2.987654321
p=1.123456789000
q=2.9876543210e+00
```

Reading a single character:

- The simplest way of reading a single character from the standard input device (Keyboard) is by using the function **getchar()** available in the header file **stdio.h**.
- The syntax of the getchar function is:

  var_name = getchar();

  where, **var_name** is a valid variable name, initially declared as chartype.
- When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to the **getchar** function.

  For example,

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

  Suppose we enter the character ‘y’ through the keyboard, ‘y’ is assigned to ch.
- Since **getchar** is a function, it is used with a set of parenthesis **()**
- Other functions that are used to read single variables are (available in **conio.h**):

  getch(): Enables the user to enter only one character through the keyboard without pressing the enter key.

  getche(): Enables the user to enter only one character through the keyboard without pressing the enter key, and will **echo** (display on monitor) the character typed in. (The last letter ‘e’ in **getche()**stands for echo)

- The **getchar** function when executed waits until any character of the keyboard is keyed in followed by pressing the ENTER (\n) key. Only after ENTER key is pressed, execution is returned to the next statement of program.
- The **getchar** function echoes (displays on monitor) the character that has been keyed in.
- We may use the **scanf()** function also to read a single character. For the above example the use of **scanf()**would be:

```
char ch;

scanf(“%c”,&ch);
```

Example:

```
#include<stdio.h>

int main()
{
  char ans;
  printf(“\n\n\tWould you like to know my name?”);
  printf(“\n\tEnter Y for YES and N for NO:”);
  ans=getchar();
  if(ans == ‘Y’ || ans == ‘y’)
         {
           printf(“\n\n\tMy name is BUSY BEE...!!!”);

           getch();
}
  else
{
           printf(“\n\n\tYou are good for nothing...!!!”);
```

```
            getch();
          }
    return 0;
    }
```

Output of the program:

```
1. Would you like to know my name? Enter Y
   for YES and N for NO: Y My name is
   BUSY BEE...!!!


2. Would you like to know my name? Enter Y
   for YES and N for NO: N

    You are good for nothing...!!!
```

Reading Character Strings:

- Suppose we want to read the string “**Handsome**” and assign it to a string variable

  name

```
                    char name[20];
                    scanf(“%10c”,name);          OR      scanf(“%s”,name);
```

- Instead of the character group %10cif %6cis used, then only first 6 characters of Handsome i.e., Handso will be assigned to name.
- Suppose for the above first **scanf** statement we enter Balaguru Swamy, only first 10 characters will be stored in name i.e., BalaguruS

Whereas with the second **scanf** statement, only Balaguru will be stored in name. (i.e., when %s is used, if white space is encountered, **scanf** will terminate)

- 'C' supports some special character groups for string variables only. Theyare

Example:

Special character groups for strings	**Meaning**
%[characters]	☐ Only the characters specified in the square brackets are permissible in the string. ☐ If any other character is tried to enter, the string is terminated by putting a NULL constant.
%[^characters]	☐ The characters specified in the square brackets after the carot (^) symbol are not permitted. ☐ The reading of the string will get terminated when one of these characters are encountered.

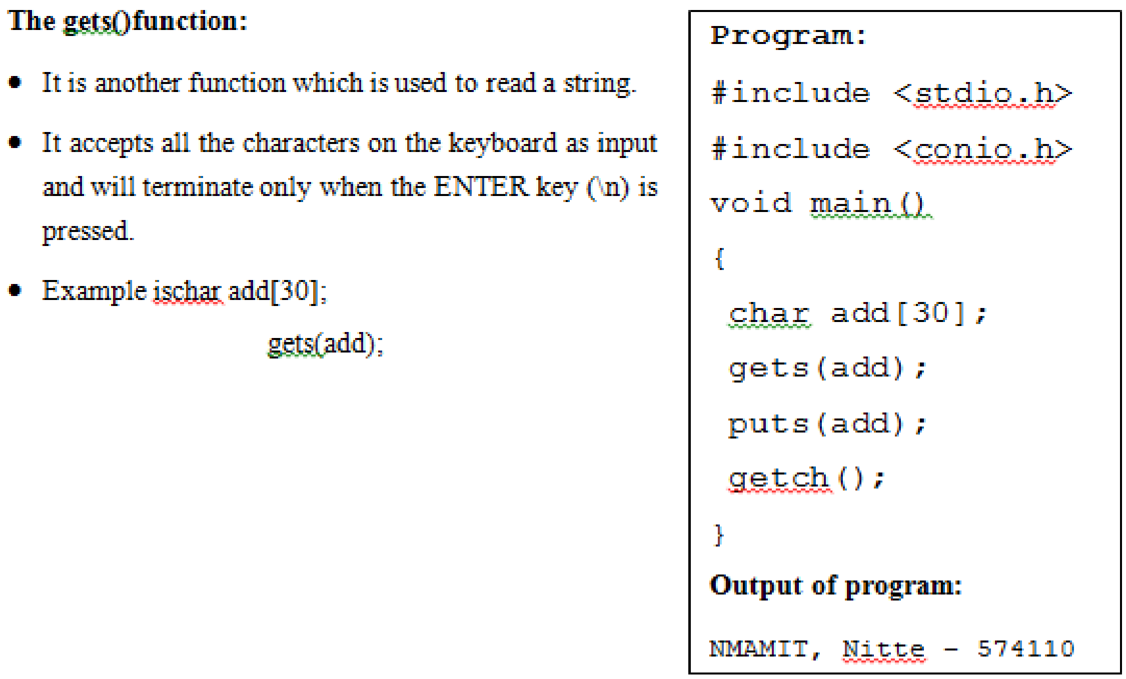
Example:

```
#include  <stdio.h>
#include <conio.h>
int main()
{
  int no;
  char name1[15],name2[15],name3[15];
  printf("\n\n\tEnter serial number                and        name1:");
  scanf("%d %15c",&no,name1);
  printf(“\n\t%d %15s”,no,name1);
  getch();

  printf(“\n\n\tEnter serial number                      and      name2:”);
  scanf(“%d %s”,&no,name2);

  printf(“\n\t%d%15s”,no,name2);
  getch();

  printf(“\n\n\tEnter serial number                      and      name3:”);
  scanf(“%d %15s”,&no,name3);

  printf(“\n\t%d %15s”,no,name3);
  return 0;

}
```

```
#include <stdio.h>
#include <conio.h>
int main()
{
char str[15];
clrscr();
printf("\n\n\tEnter address1:\n");
scanf("%[a-z]",str);
printf("\n\t%s",str);
getch();
flushall();
printf("\n\n\tEnter address2:\n");
scanf("%[^\n]",str);
printf("\n\t%s",str);
return 0;
}
```

Output of the program:

```
Enter address1:
Karkala 574110
Karkala
Enter address2:
Karkala 574110
Karkala 574110
```



The gets()function:

- It is another function which is used to read a string.
- It accepts all the characters on the keyboard as input and will terminate only when the ENTER key (\n) is pressed.
- Example ischar add[30];

  gets(add);

```
Program:
#include <stdio.h>
#include <conio.h>
void main()
{
 char add[30];
 gets(add);
 puts(add);
 getch();
}
```

Output of program:

```
NMAMIT, Nitte - 574110
```

Reading Mixed Data Types (Mixed mode Input):

- The **scanf()**function can be used to read more than one type of data (mixed mode).
- In such situations, the programmer should take care of the
  (1) **number of input variables**, (2) **data type** and (3) their **order**.
- If there is a type mismatch, **scanf()**function does not read the values input.
- For example, consider the **scanf()** statement

  scanf("%d %c %f %s",&count,&code,&ratio,name);

  The input values entered must be of the form 15 p 1.575 coffee for successful reading of the values respectively to count, code, ratio and name.

- For the above example, on successful reading, the **scanf()** function will return a value 4 (i.e., the number of data types successfully read).

- Suppose the input values to the above example are of the form:

  15   p   coffee   1.575

  - Then, **scanf()** function will return a value 2 because only two data types have been successfully read.
  - There is **match** between the control string parameters and address_list parameters only for first two input values.
  - The 3rd data was expected to be a floating type number, but a string data type has been encountered. So **scanf()** terminates the scan soon after reading the first two data types.
- If the input values for the above **scanf()**statement were of the form,

  20   150.25   motor

  The return value would be 1 since only one data (first data) has been correctly read
- Making use of the return values, we can check the correctness of data input.

```
#include <stdio.h>

int main()
{
  int a,r; float b; char c;
  printf("\n\n\tEnter the values of a, b and c:");
  r=scanf("%d %f %c",&a,&b,&c);
  if(r==3)
        printf("\n\ta=%d              b=%f      c=%c",a,b,c);
  else
        printf("\n\tERROR IN INPUT...!!");
  return 0;
}
```

Output of program:

1. Enter the values of a, b and c:
   ```
   12        3.45          A
   a=12        b=3.45        c=A
   ```
2. Enter the values of a, b and c: 23
   ```
   78 9
   a=23       b=78.000000          c=9
   ```
3. Enter the values of a, b and c: Y 12
   ```
   67
   ERROR IN INPUT...!!
   ```
4. Enter the values of a, b and c:
   ```
   15.75           23       X


   a=15       b=0.750000          c=2
   ```

FORMATTED OUTPUT

- It refers to display of the processed values or results and messages on the display screen according to the format (way of appearance) as desired by the programmer.
- It is highly desirable that the messages and values must be displayed in the most legible (understandable) form.
- The **printf()**function is a standard formatted output function defined under the header file **stdio.h**.

It’s general syntax is:

```
printf(“control string”, var_list);
                 OR
printf(“control string”, arg1,arg2,……argn);
```

- The control string indicates how many arguments follow and what their types are.

  Control string consists of three types of items:

  1. Characters that will be placed on the screen as they appear
  2. Character groups (format specifiers) that define the output format for display of each item.
  3. Escape sequence (backslash constants) such as \n, \t, \b, etc.

- **arg1, arg2,…….argn** are the list of arguments (variables) whose values are formatted and printed according to the specifications of the control string.

The general format of the character group (format specifier)used in **printf()**is

```
%w.p type_specifier
```

where,

w is an integer number that specifies the total number of columns of output value

p is another integer which specifies the number of digits to the right of the decimal point of the floating point number.

or

it is the number of characters to be printed from a string.

Both **w** and **p** are optional.

type_specifier is the character group (format specifier) like d, f, c, s, etc.

Output of integer numbers:

- The format specification for printing an integer number is

$$\boxed{\texttt{\%wd}}$$

where,

w specifies the minimum field width for output. However is a number is greater than the specified field width, it will be printed in full overriding the minimum specification.

d specifies that the value to be printed is an integer.

- The number is written right justified in the given field width.
- Leading blanks will appear as necessary.
- Preceding **w** with a minus sign (−) will cause the number to be displayed in left justification.
- Examples :

printf("%d",9876)	9	8	7	6			
printf("%6d",9876)			9	8	7	6	
printf("%2d",9876)	9	8	7	6			
printf("%-6d",9876)	9	8	7	6			
printf("%06d",9876)	0	0	9	8	7	6	

Sample Program:

```
#include <stdio.h>

int main()

{

  int m=12345;
   long n=987654;
  printf("\n\n\t%d",m);
  printf("\n\t%10d",m);
  printf("\n\t%010d",m);
  printf("\n\t%-10d",m);
  printf("\n\t%10ld",n);
  printf("\n\t%10ld",-n);

  return 0;
}
```

Output of the program:

```
12345

     12345

0000012345

12345

    987654

   -987654
```

Output of floating point numbers:

- The format specification for a floating point number in decimal notation is

%w.pf

- The format specification for a floating point number in exponential notation is

%w.pe

where,

w is an integer which specifies the minimum number of positions that are to be used for the display of the value.

p is an integer which specifies the precision i.e., the number of digits to be displayed after the decimal point.

- **w** and **p** are both optional specifications

For both the type of notations,

- The default precision is 6 decimal places.
- The negative numbers will be printed with a minus sign.
- The value when displayed, is rounded off to **p** decimal places and printed right justified in the field of **w** columns.
- Padding the leading blanks with zeros can be done by preceding **w** with a zero.
- Printing with left-justification is possible by putting a minus sign before **w**.

A special field specification character **“%*.*f”**that lets the user define the field size during run-time may also be used.

For example, printf(“%*.*f”,7,2,num);

is same as printf(“%7.2f”,num);

The advantage of this format is that the values for the **width** and **precision** may be supplied during run-time, thus making the format a dynamic one.

Example: Consider the display of the number y = 98.7654 under different format specifiers.

printf("%7.4f",y);	9	8	.	7	6	5	4				
printf("%7.2f",y);			9	8	.	7	7				
printf("%-7.2f",y);	9	8	.	7	7						
printf("%f",y);	9	8	.	7	6	5	4				
printf("%10.2e",y);			9	.	8	8	e	+	0	1	
printf("%11.4e",-y);	- 9	.	8	7	6	5	e	+	0	1	
printf("%-10.2e",y);	9 .	8	8	e	+	0	1				
printf("%e",y);	9 .	8	7	6	5	4	0	e	+	0	1

/******PRINTING OF FLOATING POINT NUMBERS******/

```
#include <stdio.h>
int main()
{
  float y = 98.7654; clrscr();
  printf("\n\n\t%7.4f",y);
  printf("\n\t%f",y);
  printf("\n\t%7.2f",y);
  printf("\n\t%-7.2f",y);
  printf("\n\t%07.2f",y);
  printf("\n\t%*.*f",10,2,y);
  printf("\n\n\t%10.2e",y);
  printf("\n\t%12.4e",-y);
  printf("\n\t%-10.2e",y);
  printf("\n\t%e",y);
  return 0;
}
```

Output of the program:

```
98.7654
98.765404
  98.77
98.77
0098.77
     98.77

 9.88e+001
-9.8765e+001
9.88e+001
9.876540e+001
```

Printing of Single Character:

- Using the formatted output, a single character can be displayed in adesired position using the format

$$\boxed{\%\, w\, c}$$

- The character will be right justified in the field of **w** columns.
- The display can be made left justified by placing a minus sign before **w**
- The default value of w is 1.

Printing of Strings:

- The format specification for printing strings is similar to that of floatingpoint numbers.

$$\boxed{\%\, w\, .\, p\, s}$$

- where **w** specifies the field width for display
- **p** specifies that only first p characters of the string are to be displayed.
- The display is right justified

Consider a string consisting of 16 characters (including blanks) "BANGALORE 560010"

Specification	**Output**																			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**
%s	B	A	N	G	A	L	O	R	E		5	6	0	0	1	0				
%20s					B	A	N	G	A	L	O	R	E		5	6	0	0	1	0
%20.10s											B	A	N	G	A	L	O	R	E	
%.5s	B	A	N	G	A															
%-20.10s	B	A	N	G	A	L	O	R	E											
%5s	B	A	N	G	A	L	O	R	E		5	6	0	0	1	0				

```
/****PRINTING OF CHARACTERS AND STRINGS****/

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

void main()
{
  char x = 'A';
          char name[25] = "MAHENDRA SINGH DHONI";

              printf("\n\n\tOUTPUT OF CHARACTERS\n");
  printf("\n\t%c\n\t%3c\n\t%5c",x,x,x);
  printf("\n\t%3c\n\t%c",x,x);
  getch();
  printf("\n");
  printf("\n\tOUTPUT OF STRINGS\n");
  printf("\n\t%s",name);
  printf("\n\t%25s",name);
  printf("\n\t%25.10s",name);
  printf("\n\t%.5s",name);
  printf("\n\t%-25.10s",name);
  printf("\n\t%5s",name);
  getch();
}
```

Output of the Program:

```
OUTPUT OF CHARACTERS

A
  A
    A
  A
A

OUTPUT OF STRINGS

MAHENDRA SINGH DHONI
     MAHENDRA SINGH DHONI
               MAHENDRA S
MAHEN
MAHENDRA S
MAHENDRA SINGH DHONI
```

Unformatted output of single character:

- Using unformatted output, a single character can be output by using the function **putchar()** available under the header file **stdio.h.**
- The syntax of **putchar()**is:

```
putchar(var_name);
```

where **var_name**is a **char** type variable containing a character.

- This statement displays the character contained in **var_name**on the display monitor.

Unformatted output of a string:

- Using unformatted output, a string can be output using the function **puts()** available under the header file **stdio.h**.
- The syntax of **puts()**function is

```
puts(string);
```

where, **string**is either a <u>variable name</u> containing a string or it is a <u>string constant</u>.

- This statement displays the string on to the display terminal and appends a new line character(**\n**) in the end.

Additional Info : **Character Testing Functions:**

'C' has a set of library functions which are used to test the type of character, i.e., whether a character is a lower case or upper case alphabet or a digit or printable character, etc.

All these functions are stored in a header file by name **ctype.h**

Some of the Character Testing Functions

Function	**Test**
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c a lower case letter?
isupper(c)	Is c an upper case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?

isspace(c)	Is c a white space character?

```
/****WRITING A CHARACTER TO SCREEN****/
#include          <stdio.h>
#include         <conio.h>
#include <ctype.h>

int main()
{
  char c;
  printf("\n\nEnter a character:\n");
  c=getchar();
  if(islower(c))
          putchar(toupper(c));
  else
          putchar(tolower(c));

          return 0;

          }
```

Output of Program:

```
1.  Enter a character:
    a
    A
2.  Enter a character:
    M
    m
```

)

Chapter 2
Decision making and Branching.

C program is a set of statements which are normally executed sequentially in the order in which they appear. This is called ***Sequential Execution***.

This happens when no options or repetitions of certain calculations are necessary.

But, in actual practice, the programmer may have to:

- change the order of execution of statements based on certain conditions
- repeat a group of statements until certain specified conditions are met

This involves a kind of ***decision making*** to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.
To handle such situations, C provides ***Control Statements*** or ***Decision Making Statements.***

The control statements determine the flow of control in aprogram.
Types of Control Statements are:
(1) Conditional Branching Control Statements
(a) **if** statement
(b) **if-else** statement
(c) **nested if-else** statement
(d) **switch** statement
(2) Unconditional Branching Control Statement
(a) **goto** statement

CONDITIONAL BRANCHING

(A) THE SIMPLE IF STATEMENT

It is a powerful decision making statement used to control the flow of execution of statements. It is called one way branching statement.
It involves a test expression (a relational or conditional expression).

Syntax and flowchart of if statement:

Single statement

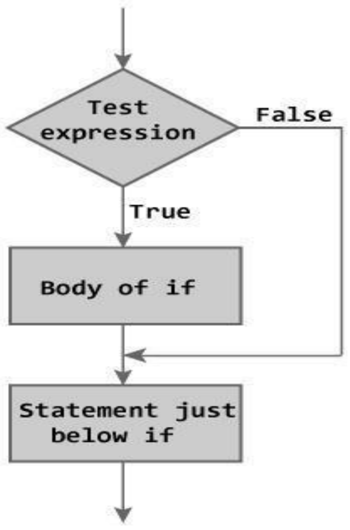
```
if (testExpression)
{
      statement;        // body of if
}
statement –x;
```

Compound Statement

```
if (testExpression)
{
     statement-1;
     statement -2;
     ………
     statement –n;
}
statement –x;
```

The statement associated with an **if** statement may be either a single statement or compound statement.

if statement evaluates the test expression inside the parenthesis.
If the test expression is evaluated to true (nonzero), statements inside the body of if is executed.
If the test expression is evaluated to false (0), a statement inside the body of if is skipped from execution.



Example (1)

```
/***USE OF IF STATEMENT  (SINGLE  STATEMENT)***/
#include<stdio.h>
int main()
{
   int number=0;
  printf("Enter a number:");
  scanf("%d",&number);
  if(number%2==0)
      {
              printf("%d is even number",number);
      }
return 0;
}
```

Output

```
Enter a number:4
4 is even number
enter a number:5
```

Example (2):

```
/***USE OF IF STATEMENT   (COMPOUND STATEMENT)***/
#include<stdio.h>
int main()
{
      int grade = 85;
       if (grade >= 60)
```

```
  {
     printf("You passed!\n");
     printf("Congratulations!\n");
     }
return 0;
}
```

Example (3): Write a C program to print the number entered by user only if the number entered is negative.

```
#include<stdio.h>
int main(
{
      int num;
      printf("Enter a number to check.\n");
      scanf("%d",&num);
      if(num<0)
            printf("Number=%d\n",num);
      printf("The if statement in C programming is easy.");
return 0;
}
```

Output 1

```
Enter a number to check.
-2
Number=-2
The if statement in C programming is easy.
```

Output 2

```
Enter a number to check.
5
The if statement in C programming is easy.
```

(B) THE IF-ELSE STATEMENT

The **if** statement provides one way branching, i.e., it executes the statement/s associated with **if**, only when the test expression is true, but does nothing if it is false.

The **if-else** statement is an extension of the **if** statement. The **if-else** statement is a *two way branching*, i.e., it executes one set of statement/s if the test expression is true or it executes another set of statement/s if the test expression is false.

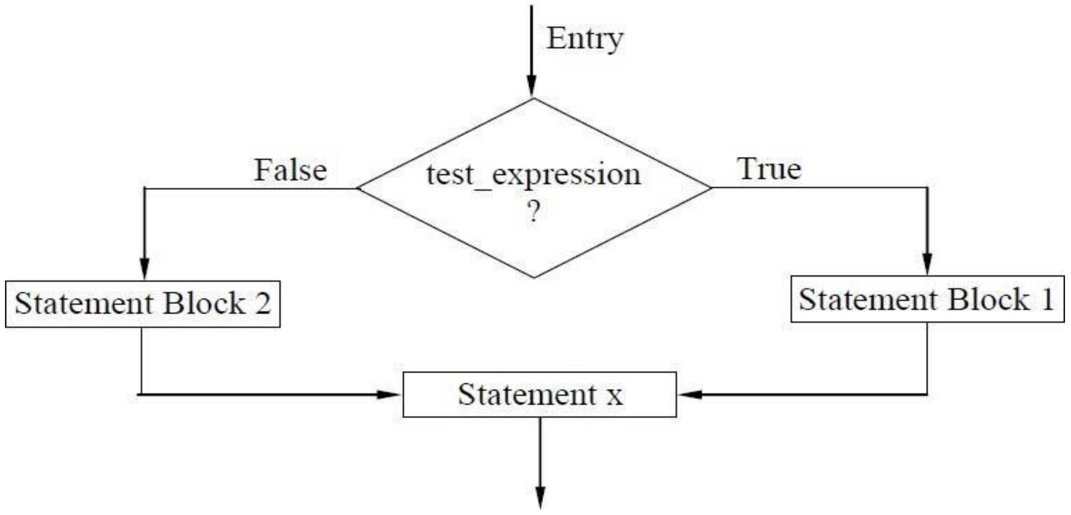
Syntax of the if-else statement

:if-else withsinglestatement

```
if(test_expression)
    statement_true;
    else
    statement_false;
    statementx;
    statementy;
```

if-else with compound statement

```
if(test_expression)
{
        statement 1;
        statement 2;
        statement 3;
}
else
{
        statement 4;
        statement 5;
        statement 6;
}
next statement;
}
```



Example: C Program to check whether a Person is eligible to Vote or not.

```
/***USE OF IF-ELSE STATEMENT  (SIMPLE  STATEMENT)***/

      #include <stdio.h>

      int main()
      {

         int age;

         printf("Enter age : ");
         scanf("%d", &age);

         if (age >= 18)
            printf("You can Vote!");
         else
            printf("You cant Vote!");

         return 0;
}
```

Output:

```
Enter age : 20
    You can Vote!
Enter age : 15
    You can‟t Vote!
```

(C) NESTED IF STATEMENTS

When a series of decisions are involved, we may have to use more than one **if-else** statement. Enclosing one **if** within another is called **nested if** statement.

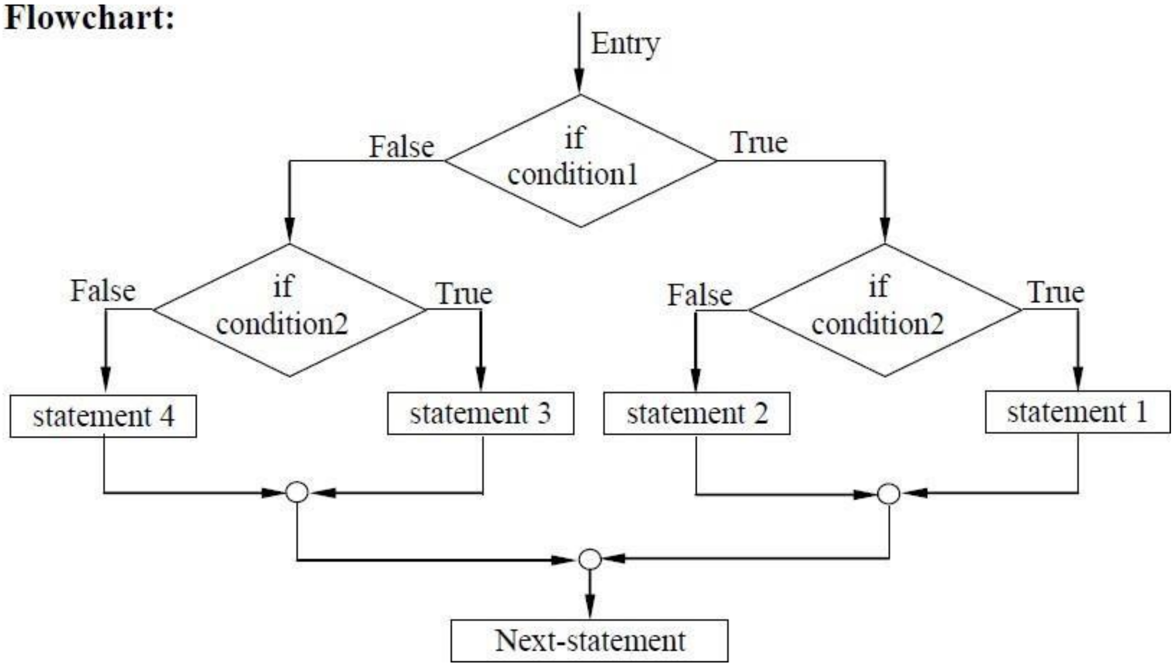
Syntax:

```
if (condition1)
{
        if (condition2)
        {
              statement1;
```

```
                                                                    }
        else
       {
               statement2;
        }
  }
else
{
  if (condition3)
{
     statement3;
 }
 else
 {
          statement4;
  }
 }
```

1. If **condition1** is TRUE and **condition2** is also TRUE, then **statement 1** is executed.
2. If **condition1** is TRUE and **condition2** is FALSE, then **statement 2** is executed.
3. If **condition1** is FALSE and **condition3** is TRUE, then **statement 3** is executed.
4. If **condition1** is FALSE and **condition3** is also FALSE, then **statement 4** is executed.

Flowchart:



Example 1: Write a C program to read values for 3 sides of a triangle, check if the triangle is equilateral, isosceles or scalene and report it.

```
/***USE OF NESTED IF***/
#include<stdio.h>
int main()
{
float s1,s2,s3;
printf("\n\nEnter the 3 sides of triangle:");
scanf("%f%f%f",&s1,&s2,&s3);
if((s1+s2)>s3&&(s2+s3)>s1&&(s3+s1)>s2)
{
printf("\n\nGiven 3 sides form a triangle...");
```

```
 if(s1==s2 && s2==s3)
printf("\nIt is an Equilateral  triangle");
else
if(s1==s2||s2==s3||s3==s1)
printf("\nIt is an isosceles triangle");
else
printf("\nIt is a scalene triangle");
 }
else
printf("\n\nSorry...entered 3 sides do not form atriangle...");
return 0;
}
```

Example 2:

```
                    /***USE OF NESTED IF***/

     #include<stdio.h>
     int main(){
         int a,b,c;
         printf("\nEnter Three Numbers : ");
         scanf("%d%d%d",&a,&b,&c);//100 98 105
         if(a>b){
             if(a>c){
                 printf("\n%d is greatest number ",a);
             }else{
                 printf("\n%d is greatest number ",c);
             }
         }else if(b>a){
             if(b>c){
                 printf("\n%d is greatest number ",b);
             }else{
                 printf("\n%d is greatest number ",c);
             }
         }else{
             printf("\n%d is greatest number ",a);
```

```
        }
        return 0;
    }
```

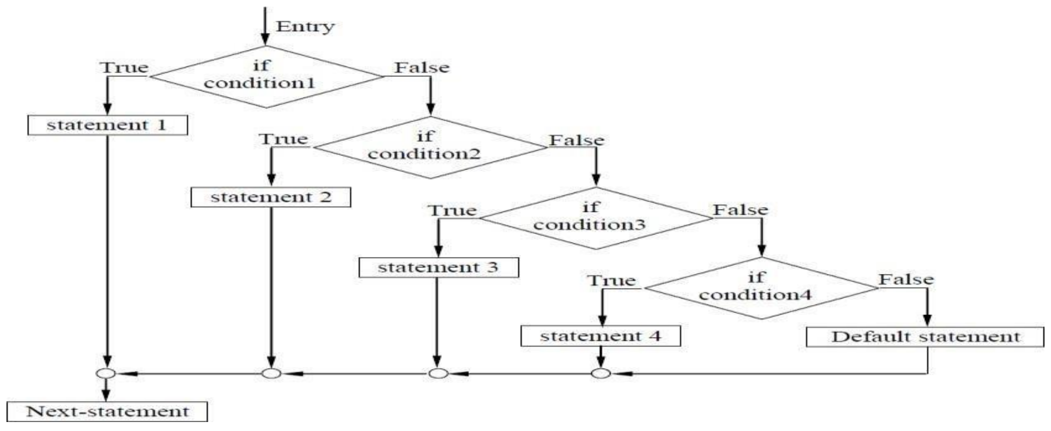
(D) THE ELSE-IF LADDER

When multipath decisions are involved, we can have a chain of **if**s in which the statements associated with each **else** is an **if**.

Syntax:

```
if(condition1)

        statement 1;
        else if(condition2)
              statement  2;
            else if(condition3)
              statement 3;
                else if(condition4)
                   statement 4;
                ----------------------

               ----------------------
                          ...........-
                   if(condition n)
                     statement n;
                   else
                     default-statement;
```

Flowchart:



Example1: Consider the problem of grading students in academic institutions.

Average marks	Grade
80-100	A
60-79	B
50-59	C
40-49	D
0-39	F

```
/***USE OF ELSE-IF LADDER***/
#include <stdio.h>
int main()
{
intmarks; char grade;
  printf("\n\nEnter average marks of students:");
 scanf("%d",&marks);
 if(marks > 79)
{
grade =  „A‟;
else if(marks > 59)
         grade = „B‟;
else if(marks > 49)
            grade = „C‟;
else if(marks > 39)
           grade = „D‟;
 else
grade = „F‟;
printf("\nGrade of the student is %c",grade);
}
return 0;
}
```

/***USE OF ELSE-IF LADDER***/

Example2: Program to relate two integers using =, > or < symbol

```
#include <stdio.h>
int main() {
   int number1, number2;
   printf("Enter two integers: ");
   scanf("%d %d", &number1, &number2);

   //checks if the two integers are equal.
   if(number1 == number2) {
      printf("Result: %d = %d",number1,number2);
   }

   //checks if number1 is greater than number2.
   else if (number1 > number2) {
      printf("Result: %d > %d", number1, number2);
   }

   //checks if both test expressions are false
   else {
      printf("Result: %d < %d",number1, number2);
   }

   return 0;
}
```

Output:

```
Enter two integers: 12    23
Result: 12 < 23


Enter two integers: 45    45
Result: 45 = 45


Enter two integers: 833     222
Result: 833 > 222
```

(D) SWITCH STATEMENT

The **nested if** allows selecting one of the many alternatives but it is time consuming because it tests all the conditions and based on the result a particular branch is taken for execution.

To overcome this disadvantage, the **switch** statement is used.
The **switch**statement provides a ***multiple way branching***.

- It allows the user to select any one of the several alternatives, depending on the value of the expression.

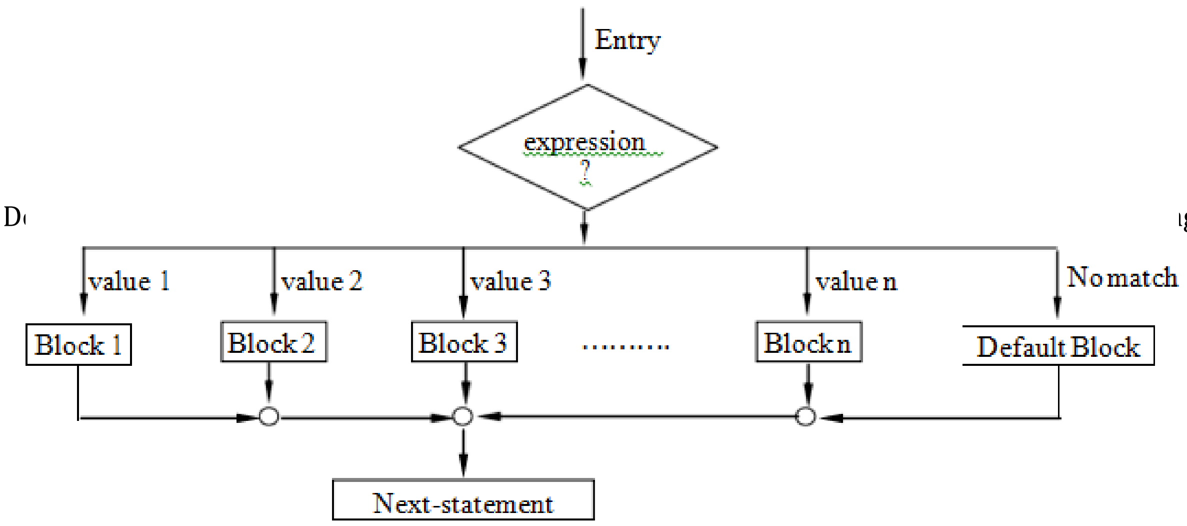
Syntax:

```
switch (expression)
{
case value1: block1;
                break;
case value2: block2;
                  break;
case value3: block3;
break;
              --------------
              --------------
              --------------
case value n:block n;
                break;
default:default block;
                break;
}
```

Note:

(1) If a break statement is omitted in switch statement, all the next cases until the very next break statement will be executed.
(2) Break statement after the default case need not be included, because, any how the control will be transferred to the very next statement after switch statement.

Flowchart:



Example 1:

```
/***USE OF SWITCH STATEMENT***/

#include <stdio.h>

int main()

{

int i;
printf("\n\nEnter i  (1 to 4):");

scanf("%d",&i);
switch(i)

{

case 1:
printf("\n\nI am in case 1...");

break;

case 2:
printf("\n\ nI am in case 2...");

break;

case 3:
printf("\n\ nI am in case 3...");

break;

case 4:
printf("\n\ nI am in case 4...");

break;

default:
printf("\n\ nI am in default case...");
}

return 0;

}
```

Example 2: ***/***USE OF SWITCH STATEMENT***/***

```
#include <stdio.h>
int main()
{
float len,br,ht,base,side,radius;
char shape;
printf("\n\nEnter your choice:");
printf("\nEnter „c‟ to compute area of circle:");
printf("\nEnter „s‟ to compute area of square:");
printf("\nEnter „r‟ to compute area of rectangle:");
printf("\nEnter „t‟ to compute area of triangle:");
scanf("%c",&shape);
switch(shape)
{
case „c‟:
case „C‟:
printf("\n\Area of a Circle...");
printf("\nEnter the radius of the circle:");
scanf("%f",&radius);
area = 3.142*radius*radius;
printf("\n\nArea of a Circle is %f",area);
break;

case „s‟:
case „S‟:
printf("\n\nArea of a Square...");
printf("\nEnter the side of the square:");
scanf("%f",&side);
area = side*side;
printf("\n\nArea of a Square is %f",area);
break;

case „r‟:
case „R‟:
printf("\n\nArea of a Rectangle...");
printf("\nEnter length and breadth of the rectangle:");
scanf("%f%f ",&len,&br);
area = len*br;
printf("\n\nArea of Rectangle is %f",area);
break;
case „t‟:
case „T‟:
printf("\n\nArea of a Triangle...");
printf("\nEnter base and altitude of the triangle:");
scanf("%f  %f ",&base,&ht);
area = 0.5*base*ht;
printf("\n\nArea of Triangle is %f",area);
break;

default:
printf("\n\nSorry...Unable to identify your choice");
}
return 0;
 }
```

Example 3: Write a program in C using switch statement to device a Basic Arithmetic Calculator.

```
/***USE OF SWITCH STATEMENT***/
#include <stdio.h>

int main()
{
float n1,n2; char oper;
printf("\n\nEnter the two numbers which must be operated:");
scanf("%f%f",&n1,&n2);
printf("\n\n Select the operation you want to perform:");
printf("\nEnter „A‟ to compute SUM:");
printf("\nEnter „S‟ to compute DIFFERENCE:");
printf("\nEnter „M‟ to compute PRODUCT:");
printf("\nEnter „D‟ to compute QUOTIENT:");
scanf("%c",&oper);
switch(oper)
{
case „a‟:
case „A‟:
printf("\n\nPerforming addition...");

printf("\n\nThe sum of %f & %f is %f",n1,n2,n1+n2);
break;

case „s‟:
case „S‟:
printf("\n\nPerforming    subtraction...");
printf("\n\nThe difference of %f & %f is %f",n1,n2,n1-n2);
break;

case „m‟:
case „M‟:
printf("\n\nPerforming  multiplication...");
printf("\n\nThe product of %f & %f is %f",n1,n2,n1*n2);
break;

case „d‟:
case „D‟:
printf("\n\nPerforming    division...");
printf("\n\nThe quotient of %f & %f is %f",n1,n2,n1/n2);
break;

default:
printf("\n\nSorry...Unable to identify your choice");
}
return 0;
}
```

UNCONDITIONAL BRANCHING:

THE GOTO STATEMENT

The conditional control statements control the flow of execution of statements based on the value of the conditional expression.

- The **goto** statement is an unconditional statement in C which allows the flow of execution of statements to be altered without the use of any conditional pression.

- The **goto** statement transfers the control from one point to another in a C program. This is also known as *Jumping of control* or *Unconditional Jumping*.

The syntax of **goto**statement is as follows:

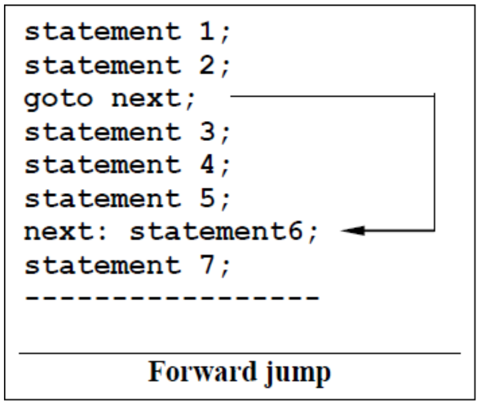
goto label;

where, **goto** is a keyword, and
label is a symbolic constant or an identifier which need not be pre-declared.

- Two types of jumping (branching) is possible by using **goto** statement:

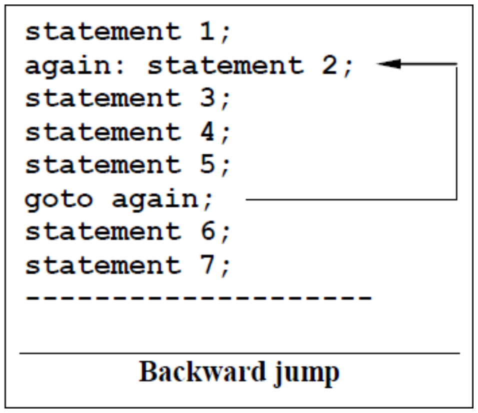
  (1) Forward jump: Here the statements immediately following the **goto** statement will be skipped and control is transferred to the statement beginning with the symbolic constant **lable**.

  (2) Backward jump: Here the symbolic constant **lable** is placed before the **goto** statement and the statements between the **lable** and **goto** statements are repeated resulting in looping.



```
statement 1;
statement 2;
goto next;
statement 3;
statement 4;
statement 5;
next: statement6;
statement 7;
-----------------
```

Forward jump



```
statement 1;
again: statement 2;
statement 3;
statement 4;
statement 5;
goto again;
statement 6;
statement 7;
--------------------
```

Backward jump

Example 1: Write a program in C using goto statement to read 5 numbers and compute their sum.

```
/***USE OF GOTO STATEMENT***/
#include <stdio.h>
int main()
{
int count = 0;
float sum = 0.0, num;
printf("\n\nEnter five numbers:");
read: scanf("%f",&num);
         sum += num;
         count ++;
         if (count < 5)
        goto read;
        printf("\n        The sum of 5 numbers is %f", sum);
return 0;
}
```

Example 2: Write a program in C using goto statement to accept the number and reverses it.

```
#include <stdio.h>
int main()
{
int number, rev = 0, digit, temp_num;
printf("\nEnter a number:\n");
scanf("%d",&number);
temp_num= number;
START:
            digit = number%10;
            rev = rev*10+digit;
            number /=10;
            if (number > 0)
           goto START;
printf("\nInput number =%d\n",     temp_num);
printf("\nReversed number =%d\n",rev);
return 0;
}
```

The ?: operator

The **?:** operator is popularly known as *conditional operator*, useful for making *two–way* decisions and takes *three* operands.

The general form of the use of conditional operator is as follows:

Conditional_expression? expression1 : expression2

- The **conditional_expression** is evaluated first. If the result is true (nonzero),

expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, **expression2** is evaluated and its value is returned.

Example 1:
```
int a = 10, b = 20, c;

        c = (a < b) ? a : b;

        printf("%d", c);
```
output:10

Example 2:
```
#include <stdio.h>

int main()
{
   int age;

 // take input from users
   printf("Enter your age: ");
   scanf("%d", &age);

   // ternary operator to find if a person can vote or not
   (age >= 18) ? printf("You can vote") : printf("You cannot vote");

 return 0;
}
```
Output :
```
Enter your age: 12
You cannot vote
```
Example 3:

```
#include <stdio.h>

int main() {

  // create variables
  char operator = '+';
  int num1 = 8;
  int num2 = 7;

  // using variables in ternary operator
  int result = (operator == '+') ? (num1 + num2) : (num1 - num2);
  printf("%d", result);

  return 0;
}

// Output: 15
```

Example 4:

```
int a = 1, b = 2, ans;
                    if (a == 1)
                              {
                              if (b == 2)
                               {
                                        ans = 3;
                                                 } else {
                                                           ans = 5;
                                                           }
                              } else {
                                        ans = 0;
                                        }
```

```
                printf ("%d\n", ans);
```

Using Conditional Operator, the above code can be written as:

```
int a = 1, b = 2, ans;
ans = (a == 1 ? (b == 2 ? 3 : 5) : 0);
printf ("%d\n", ans);
```

Output: 3

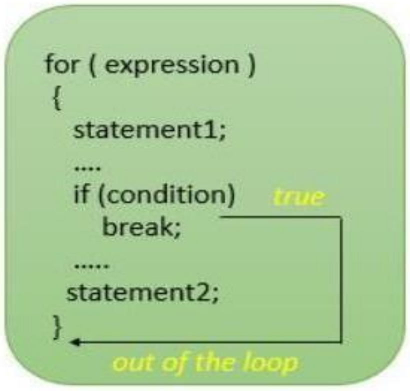
THE BREAK STATEMENT

In any loop break is used to jump out of loop skipping the code below it without caring about the test condition.

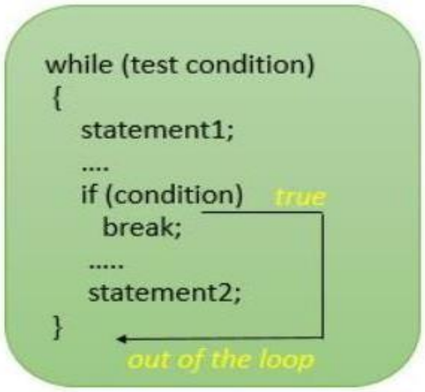
It interrupts the flow of the program by breaking the loop and continues the execution of code which is outside the loop.

The common use of break statement is in switch case where it is used to skip remaining part of the code.

Structure of Break statement



```
for ( expression )
{
   statement1;
   ....
   if (condition)
      break;
   .....
  statement2;
}
```



```
while (test condition)
{
   statement1;
   ....
   if (condition)
     break;
   .....
   statement2;
}
```

Example: C program to take input from the user until he/she enters zero.

```
#include <stdio.h>
int main ()
```

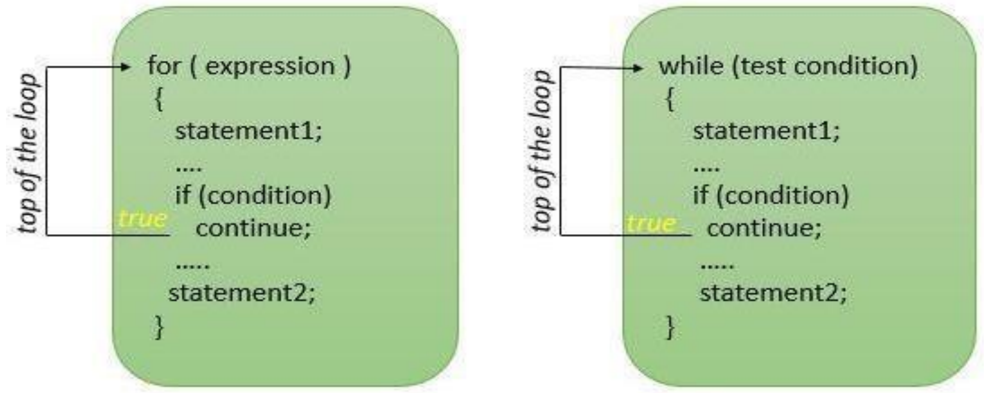
```
{
 int a;
 while (1)
 {
   printf("enter the number:");
   scanf("%d", &a);
   if ( a == 0 )
      break;
  }
 return 0;
}
```

In above program, while is an infinite loop which will be repeated forever and there is no exit from the loop.
So the program will ask for input repeatedly until the user will input 0.
When the user enters zero, the if condition will be true and the compiler will encounter the break statement which will cause the flow of execution to jump out of the loop.

THE CONTINUE STATEMENT

Like a break statement, continue statement is also used with if condition inside the loop to alter the flow of control.
When used in while, for or do...while loop, it skips the remaining statements in the body of that loop and performs the next iteration of the loop.
Unlike break statement, continue statement when encountered doesn‟t terminate the loop, rather interrupts a particular iteration.

Structure of Continue statement



```
for ( expression )
{
   statement1;
   ....
   if (condition)
      continue;
   .....
  statement2;
}
```

```
while (test condition)
{
   statement1;
   ....
   if (condition)
     continue;
    .....
    statement2;
}
```

Example: C program to print sum of odd numbers between 0 and 10

```
#include <stdio.h>
int main ()
{
 int a,sum = 0;
 for (a = 0; a < 10; a++)
 {
      if ( a % 2 == 0 )
      continue;
  sum = sum + a;
  }
 printf("sum = %d",sum);
 return 0;
}
```

Output

```
sum = 25
```

Chapter 3
Decision making and Looping.

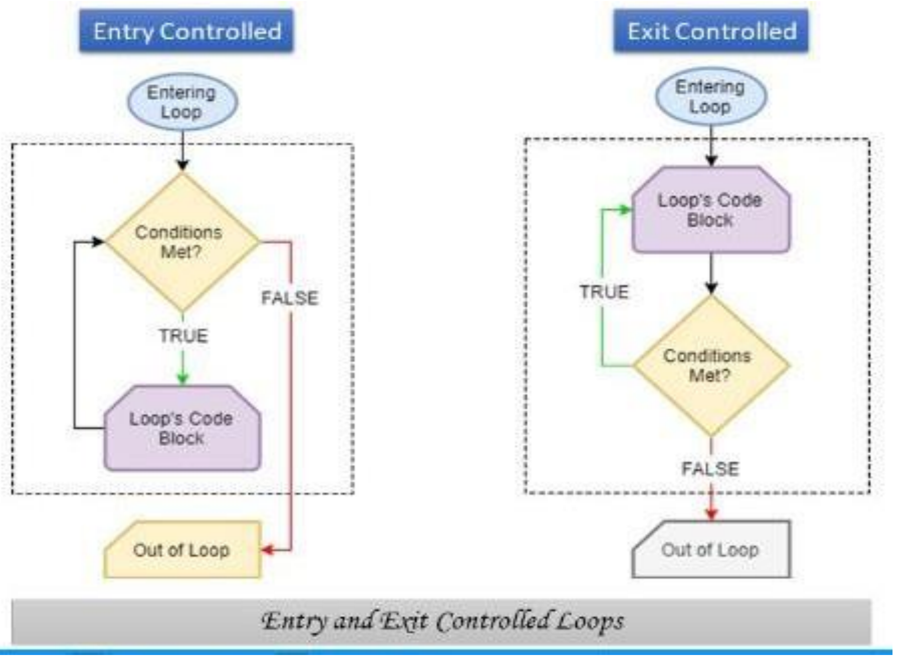
Looping is a mechanism by means of which a set of statements are executed repeatedly for a fixed number of times or until a condition is fulfilled.

Looping is also called ***repetitive*** or ***iterative*** control mechanism. A program loop consists of ***two*** segments.

- Body of the loop.
- Control statement.
-

Control statement tests certain conditions and directs the repeated execution of the statements contained in the *body of the loop*.

Depending on the position of the *control statement* in the loop, a control structure is classified as ***entry – controlled loop*** and ***exit – controlled loop***.

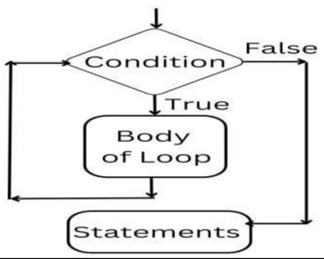
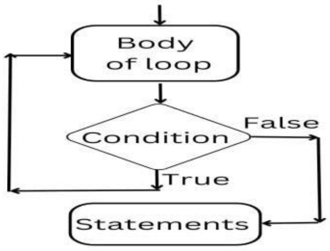


Loop control structures

- In ***entry – controlled loop***, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed.

- In ***exit – controlled loop***, the test is performed at the end of the loop and therefore the body is executed unconditionally for the first time.

Differences Between entry controlled loop and Exit Controlled loop

	Entry Controlled loop	Exit Controlled loop
1	Pre-test loop	Post-test loop
2	check the condition before entering the loop	check the condition after executing the loop
3	Body of the Loop will be executed zero times.	Body of the Loop will be executed at least one time.
4		

5	#include<stdio.h>
	int main()
	{
	int i=10;
	while(i<10)
	{
	printf("I will not be executed as it is entry controlled loop");
	i++;
	}
	return 0;}
		#include<stdio.h>
		int main()
		{
		int i=10;
		do
		{
		printf("I will be executed at once as it is exit controlled loop");
		i++;
		}while(i<10);
		return 0;
		}

6	Ex:-For,while loop	Ex:-Do while loop

Steps of efficient looping mechanism

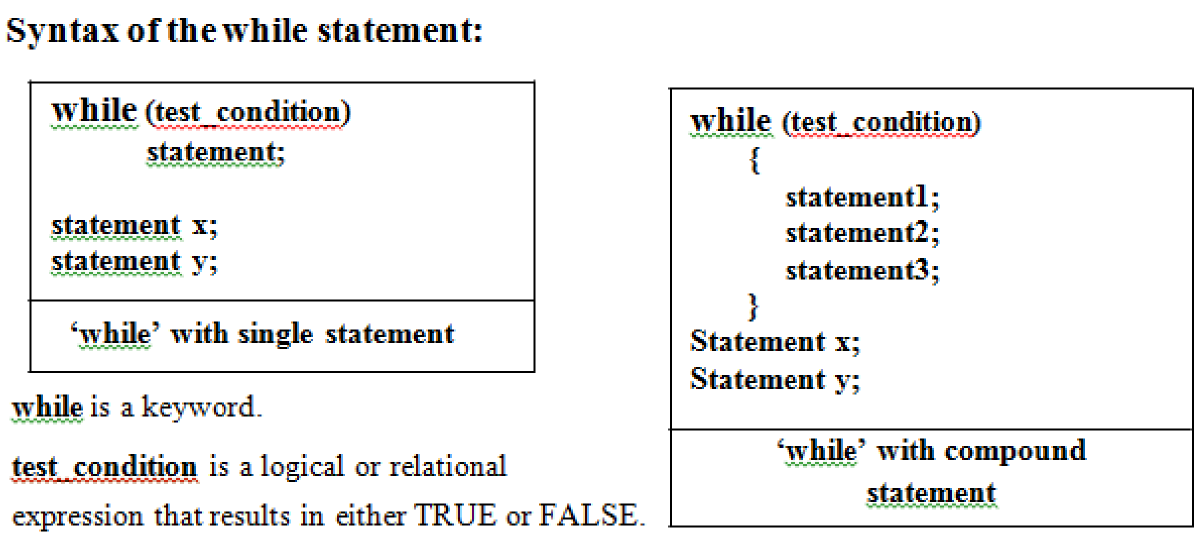
- **Initialization:** To set the initial value for the loop counter; it may be an increment or decrement counter.
- **Decision:** An appropriate **test_condition** to determine whether the loop should be executed or not and eventually should allow the loop to terminate. Otherwise it may form an *infinite loop*.
- **Body of the loop:** To execute the body of the loop when the *test condition* is TRUE.
- **Updating loop counter:** Increment or decrementing the loop counter.

C language provides ***three*** types of loop structures.

- **while** statement.
- **do while** statement.
- **for** statement.

(A) THE WHILE STATEMENT

- It executes a set of statements repeatedly as long as the specified condition is true.



Syntax of the while statement:

```
while (test_condition)
          statement;

statement x;
statement y;
```

'while' with single statement

while is a keyword.

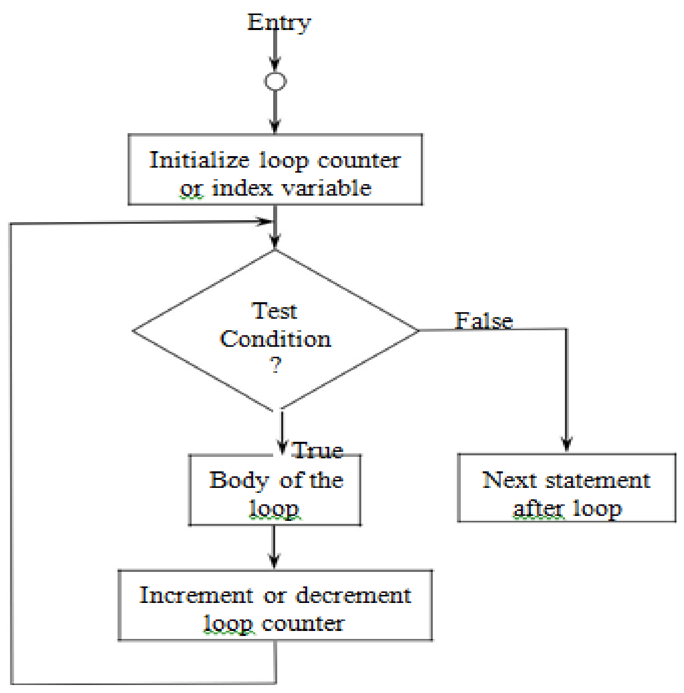
test_condition is a logical or relational expression that results in either TRUE or FALSE.

```
while (test_condition)
      {
          statement1;
          statement2;
          statement3;
      }
Statement x;
Statement y;
```

'while' with compound statement

- The **while** is an *entry – controlled loop* statement.
- The *test condition* is evaluated and if the condition is *true*, then the body of the loop is *repeatedly* executed.
- If the condition is *false*, then control comes out of the loop and continues with next executable statement.
- The value of the *test condition* will change during every pass of while loop. If it is not changed, then while loop enters an infinite loop (i.e. control does not come out).

Flowchart:


Entry
Initialize loop counter or index variable
Test Condition ?
False
True
Body of the loop
Next statement after loop
Increment or decrement loop counter

Example (1): Using while statement, write a C program to accept your name and prints it desired number of times.

```
/***USE OF while STATEMENT***/
#include <stdio.h>

int main()
   {
     char name[20];
     int ktimes, i;
     printf("\nWhat is your name?\n");
     gets(name);
     printf("\nHow many times to print your name\n");
     scanf("%d",&ktimes);
     i=0;

     while(i < ktimes)
```

```
  {
        puts(na
        me);i++;
  }
return 0;
}
```

Output of program:

```
What is your name?

NMAMIT

How many times to print your name 3

NMAMIT
NMAMIT

NMAMIT
```

2. // Print numbers from 1 to 5

```c
#include <stdio.h>
int main()
{
  int i = 1;

  while (i <= 5)
 {
    printf("%d\n", i);
    ++i;
  }

  return 0;
}
```

Output

```
1
```

```
2
3
4
5
```

Example (3): Using while statement, write a C program to compute **x** to the power **n**. where n isnon negative number.

```
/***USE OF while
STATEMENT***/#include
<stdio.h>
int main()
    {
      int
      count, n;
      float x,y;

      printf("\nEnter the values of x and
      n:");scanf("%f %d",&x, &n);
      y=1.0;

      count=1;
```

```
  while(count <= n)
  {
        y = y*x;
        count++;
  }
  printf(“x = %f; n = %d; x to the power n = %f”, x, n, y);
  return 0;
}
```

Output of program:

```
Enter the values of x and n: 2.5 4

 x = 2.500000; n = 4; x to the power n = 39.062500

 Enter the values of x and n: 0.5 4

 x = 0.500000; n = 4; x to the power n = 0.062500
```

Example 4: Program to print table for the given number using while loop in C

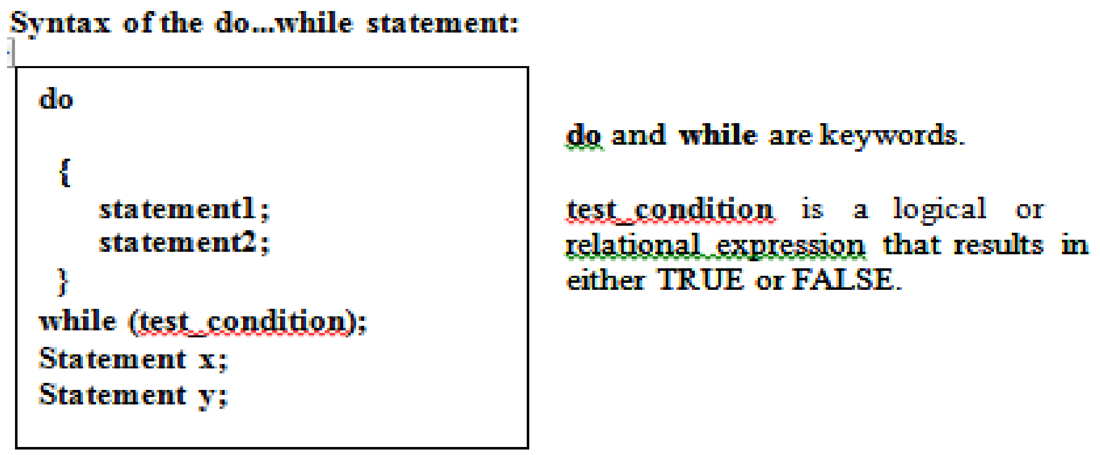
```
#include<stdio.h>
int main()
{
int i=1,number=0,b=9;
printf("Enter a number: ");
scanf("%d",&number);
      while(i<=10)
      {
      printf("%d \n",(number*i));
      i++;
      }
return 0;
}
```

Output

```
Enter a number: 50
50
100
```

```
150
200
250
300
350
400
450
500
```

(B) *THE DO WHILE STATEMENT*



Syntax of the do...while statement:

```
do

  {
     statement1;
     statement2;
  }
while (test_condition);
Statement x;
Statement y;
```

do and **while** are keywords.

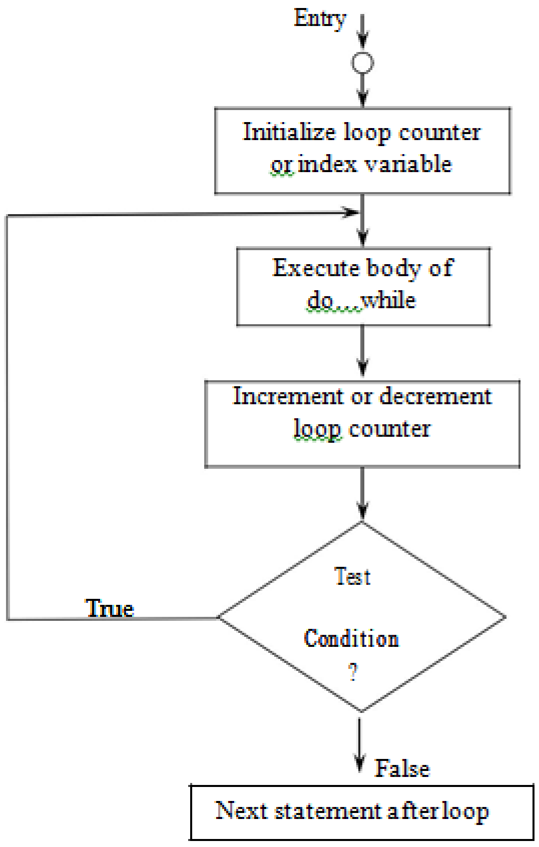
test_condition is a logical or relational expression that results in either TRUE or FALSE.

- On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test_condition in the **while** statement is evaluated.
- If the condition is *true*, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true.

 When the condition becomes *false*, the *loop* will be terminated and the control goes to the statement that appears immediately after the while statement.

 Since the test_condition is evaluated at the bottom of the loop, the **do...while** construct provides an ***exit controlled*** loop and therefore the body of the loop is *always executed at least once*.

Flowchart:


Entry
Initialize loop counter or index variable
Execute body of do....while
Increment or decrement loop counter
Test Condition ?
True
False
Next statement after loop

Example(1) prints numbers from 1 to 5:

```
#include <stdio.h>
int main()
{
inti = 1;
        do {
        printf("%d\n", i);
        i++;
        } while (i<= 5);
return 0;
}
```

Output:

```
1
2
3
4
5
```

Example (2): Using do..while, write a C program to print the sum of all odd integers between 1 and50.

```
/***USE OF do...while STATEMENT***/
#include <stdio.h>
int main()
{
       int odnum, sum = 0;

       odnum = 1;
         do
         {

             sum += odnum;
             odnum += 2;
         }
         while(odnum <= 50);
       printf("Sum =%d\n",sum);
       return 0;
}
```

Example 3: program for finding the average of first N natural numbers using a do...while() loop in C.

```
#include <stdio.h>

int main() {
    int sum = 0, N, i = 1;
    float avg;

    printf("Enter the value of N : ");
    // input N
    scanf("%d", &N);

    do {
        // loop body
        sum += i;

        // update expression
        i++;
    } while (i <= N);

    printf("Sum : %d", sum);

    // Average of first N numbers
    // typecasting sum from int to float data type
    avg = (float)sum / N;


    // %0.2f will print avg with a precision of 2 decimal places
    printf("\nAverage of %d numbers : %0.2f", N, avg);

    return 0;
}
```

Input

```
Enter the value of N : 12
```

Output :

```
Sum : 78
Average of 12 numbers : 6.50
```

(C) THE FOR STATEMENT

The **for** is an *entry – controlled loop* statement. This statement is used when the programmer knows how many times a set of statements are executed.

Syntax of the *simple* for statement:

```
for (initialization; test_condition; increment)
{
    statement1;
    statement2;
}
Statement x;
Statement y;
```

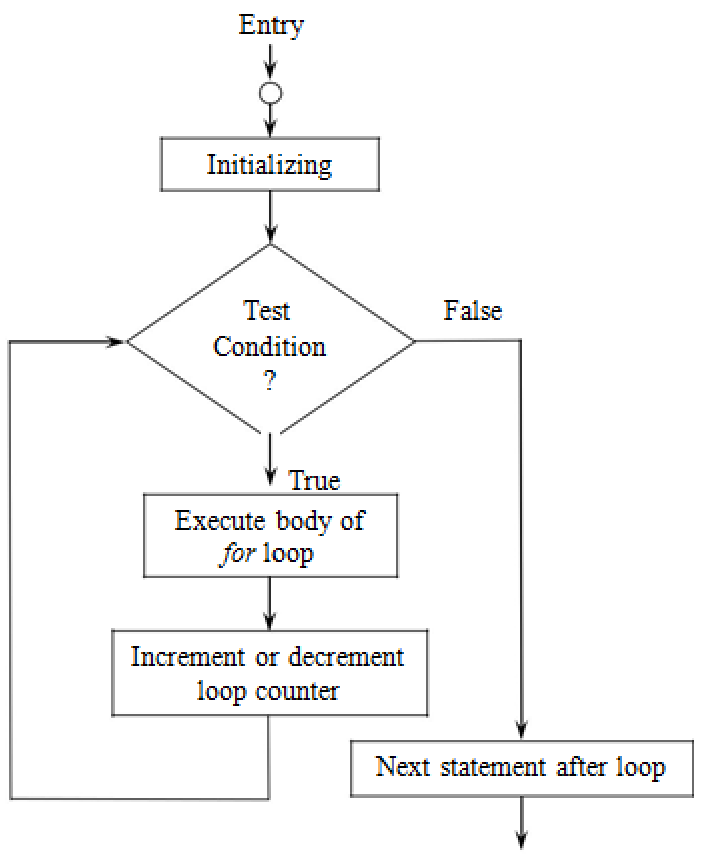
The execution of the **for** statement is as follows:

- **Initialisation** of the *control variable* is done first. Using assignment statements such as i = 1 and count = 0. The variables *i* and *count* are known as *loop – control variables.*
- The value of the *control variable* is tested using the **test_condition**. The *test_condition* is a *relational expression*, such as i < 10 that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement the immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the *control variable* is **incremented** using an assignment statement such as i = i+1 and the new value of the *control variable* is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test_condition.
- The **for** statement allows for *negative* **increment** also.
- The three sections enclosed within parentheses must be separated by ***semicolons***. There is ***no semicolon*** at the end of the **increment** section as well as after the parentheses.
- *Braces* are optional when the body of the loop contains only *one* statement.

The number of *iterations* required to execute the body of **for** is computed using the formula:

Number of iterations = (Final value – initial value + step increment) / step increment

Flowchart:


Entry
Initializing
Test Condition ?
False
True
Execute body of *for* loop
Increment or decrement loop counter
Next statement after loop

Example (1): Using *for* statement, write a C program to print the sum of first 50 natural numbers.

```
/***USE OF for STATEMENT***/
int main()
{
for(num=1; num<=50; num++)
{
sum = sum + num;
}
printf("Sum = %d\n", sum);
return 0;
}
```

Example (2): Using *for* statement, write a C program to print factorial of a given number.

Hint: The formula to compute factorial of a number **n** is

*n! = n * (n-1) * (n-2) * (n-3) *........... * 3 * 2 * 1*

```
#include <stdio.h>

int main()
{
  int c, n, f = 1;

  printf("Enter a number to calculate its factorial\n");
  scanf("%d", &n);
  for (c = 1; c <= n; c++)
    f = f * c;
  printf("Factorial of %d = %d\n", n, f);
  return 0;
}
```

Example (3): Using for statement, write a C program to generate the first n Fibonacci numbers. In fibonacci sequence each number is generated by adding the previous two numbers. Assuming the first two numbers as 0 and 1. Then the series,

0 1 1 2 3 5 8 13 …

```
/***Program to generate the first n fibonacci numbers***/
#include <stdio.h>
int main()
{
     int n, n1, n2, n3;
printf("\n Enter the limits of fibonacci series:\n");
scanf("%d", &n);
printf("\n Enter the first & second numbers in fibonacci series:\n");
scanf("%d %d", &n1, &n2);
printf(\n%d %d", n1, n2);
for(i = 3; i <= n; i++)
{
   n3 = n1 + n2;
   printf("\t%d", n3);
       n1=n2;
       n2=n3;
 }
return 0;
}
```

Additional features of* for *loop:

- More than *one variable* can be **initialized** at a time in the **for** statement, separated by a ***comma operator***.

for(p=1, n=0; n < 17; ++n)

- The **increment** section may also have more than one part.

for(n=1, m=50; n <= m; n++, m--)

- The **test_condition** may have *any compound relation* and the testing need not be limited only to the loop control variable.

```
sum = 0;
for(i=1; i < 20 && sum < 100; ++i)
     {
        sum = sum + i;
        printf("%d", sum);
        }
```

- It is also permissible to use *expressions* in the *assignment* statements of **initialization** and **increment** sections.

```
for(x = (m + n)/2; x > 0; x = x/2)
```

- In **for** statement one or more sections can be omitted, if necessary. In such cases, the sections are left blank. However, the *semicolon* separating the sections must remain.

```
m = 5;
for(; m != 100;)
 {
        printf("%d", sum);
        m = m + 5;
 }
```

- If the **test_condition** is not present, the **for** statement sets up an *infinite loop*. Such loops can be broken using **break** or **goto** statements in the loop.

- We can set up *time delay loops* using the *null* statement as follows: This loop is executed 1000 times without producing any output; it simply causes a time delay. The *body of the loop* contains *only a semicolon*, known as *null* statement.

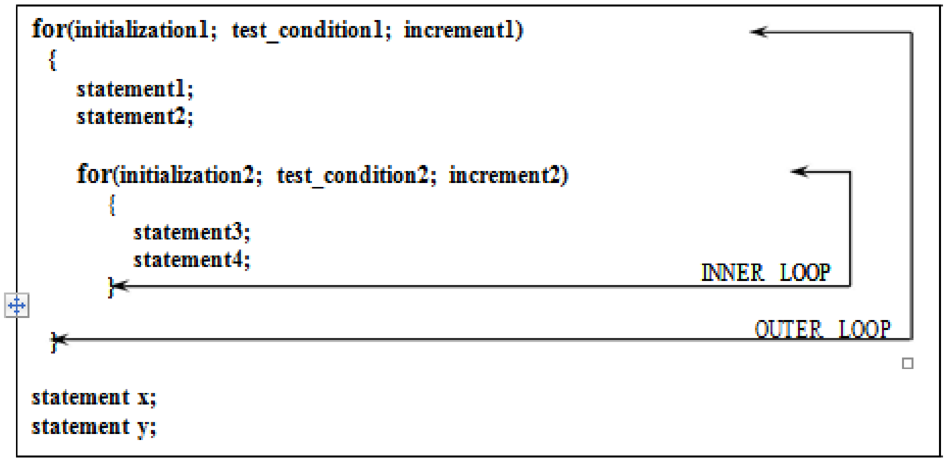
```
for(j = 1000; j > 0; j = j - 1)

;
```

- This can also be written as **for(j = 1000; j > 0; j = j - 1);** This implies that the C compiler will not give an error message if we place a *semicolon* by mistake at the end of a **for** statement. The *semicolon* will be considered as a *null statement* and the program may produce *some nonsense*.

Nesting of* for *loops:

- If one **for** *statement* is completely placed within the other, it is known as **nested for** *statement*.
- Each **for** *statement* must have *different index variable*.
- Nesting may be up to 15 levels in ANSI C; many compilers allow more.

Syntax of the *nested* for statement:



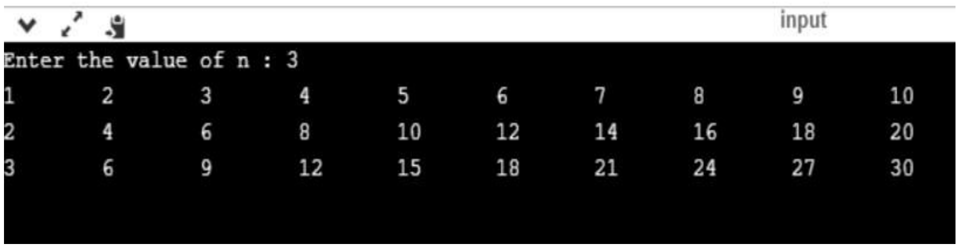
```
for(initialization1;  test_condition1;  increment1)
  {
     statement1;
     statement2;

     for(initialization2;  test_condition2;  increment2)
        {
           statement3;
           statement4;
        }
  }

statement x;
statement y;
```

// Displaying the n tables.

Example of nested for loop

```
#include <stdio.h>
int main()
{
  int n;// variable declaration
  printf("Enter the value of n :");
  // Displaying the n tables.
  for(int i=1;i<=n;i++)  // outer loop
  {
     for(int j=1;j<=10;j++)  // inner loop
     {
        printf("%d\t",(i*j)); // printing the value.
     }
     printf("\n");
  }
```

Output:



```
Enter the value of n : 3
1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30
```

Example:

A class of **n** students take an annual examination in **m** subjects. Write a C program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them.

```
/***ILLUSTRATION OF NESTED LOOPS***/
#include <stdio.h>
#define FIRST 360
#define SECOND 240
int main()
   {
     int i, j, n, m, roll_number, marks, total; clrscr ();
     printf(“Enter the number of students and subjects\n”);
     scanf(“%d %d”, &n, &m);
     printf(“\n”);

     for(i = 1; i <= n; ++i)
       {
           printf(“Enter Roll Number:”);
           scanf(“%d”, &roll_number);
           total = 0;

           printf(“\nEnter marks of %d subjects for ROLL NO %d \n”, m, roll_number);
           for(j = 1; j <= m; j++)
             {
                 scanf(“%d”, &marks);
                 total = total + marks;
             }
           printf(“TOTAL MARKS = %d”, total);

           if (total >= FIRST) printf(“(First Division)\n\n”);
             else if (total >= SECOND)
                  printf(“(Second Division)\n\n”);
               else
                 printf(“(*** FAIL ***)\n\n”);
       return 0;
       }
```

Example: Write a C program to generate the following number pattern

```
            1
         2  3  2
      3  4  5  4  3
   4  5  6  7  6  5  4
5  6  7  8  9  8  7  6  5
```

and so on.

```
#include <stdio.h>
int main()
   {
     int i, j, k=1, n, p;
     printf("Enter the number of rows\n");
     scanf("%d", &n);
     for(i = 1; i <= n; i++)
       {
           for(j = 1; j <= (n-i); j++)
             {
                 printf(" ");
             }
           for(j = 1; j <= i; j++)
             {

                 printf("%d", k++);
             }
           for(j = (i-1); j >= 1; j--)
             {
                p = k – 2;
                k = k - 1;

                printf(“%d”, p);
            }
        printf(“\n”);
    }
    return 0;
}
```

JUMPS IN LOOPS

- When executing the body of loop, sometimes it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. *For example, in the case of searching for a particular name in a list containing, say, 100 names, program loop must be terminated as soon as the desired name is found.*
- C permits a jump from one statement to another *within a loop* as well as a *jump out of a loop*.

Jumping Out of a loop:

- An early exit from a loop can be accomplished by using the **break** statement or the **goto** statement.
- **break** statement or the **goto** statement can be used within if …else, switch, while, do …while, or for loops.
- When the **break** statement is encountered *inside the loop*, the loop is immediately exited and program continues with the statement immediately following the loop.
- When the loops are *nested*, the **break** would only exit from the loop containing it. That is **break** will *exit only a single loop*.

- Since a **goto** statement can transfer the control to any place in the program, it is useful to provide *branching* within a loop.

- Important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

- But it is good practice to ***avoid*** using **goto** statement, because many compilers generate *less efficient code* if **goto** is used, and also using many **goto** statements makes *program logic complicated* and renders the program unreadable.

Example (1): Write a C program to find whether given number is prime number.

```
/***Program to find if a given number is prime***/
#include <stdio.h>
int main()
{
      int i, num, flag=0;
      printf("\n Enter the number to be checked as prime:\n");
      scanf("%d", &num);

      for(i = 2; i <= (num - 1); i++)
        {

              if (num % i == 0)
                {

                      flag = 1;
                      break;
                }

              if (flag != 1)
                printf("\n Number Entered %d is a Prime", num);

              else

                printf("\n Number Entered %d is not a Prime", num);
        }
        return 0;
    }
```

Output of program:

```
(1) Enter the number to be checked as prime: 3
        Number Entered 3 is a Prime

    (2)Enter the number to be checked as prime: 10
        Number Entered 10 is not a Prime
```

Example (2): Write a C program to list prime numbers between 1 & 300.

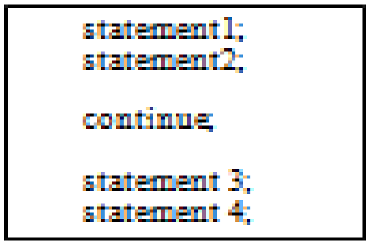
```
#include <stdio.h>

int main()
{
     int i, num, flag, np = 0;
     printf("\n List of Prime numbers between 1 & 300\n");

     for(num = 1; num <= 300; num++)
       {
           flag = 0;

           for(i = 2; i <= (num – 1); i++)
             {
                 if ((num % i) == 0)
                   {

                       flag = 1;
                       break;
                   }

                 if (flag != 1)
                   {
                       printf("%3d \t", num);

                   }

                 np = np + 1;
             }

           printf("\n Number of Prime numbers between 1 & 300 = %d", np);
```

```
        }
        return 0;
}
```

<u>Skipping a part of a loop:</u>

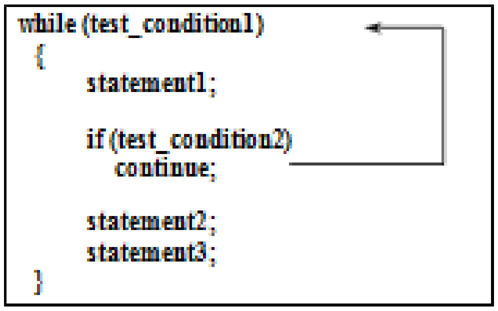
- To skip certain statements in the loop and to continue with the next iteration of the loop **continue** statement is used.
- **continue** statement is used as a bypasser.
- Control does not come out of the loop instead skips the remaining statements within the body of that loop and transfer to the beginning of the loop.

Syntax of the continue statement:

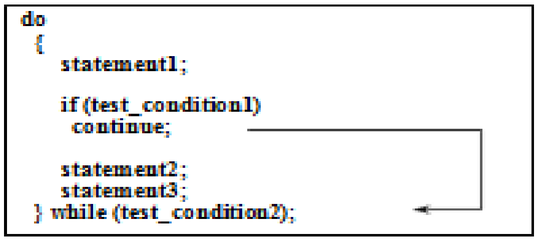


```
statement1;
statement2;

continue;

statement 3;
statement 4;
```

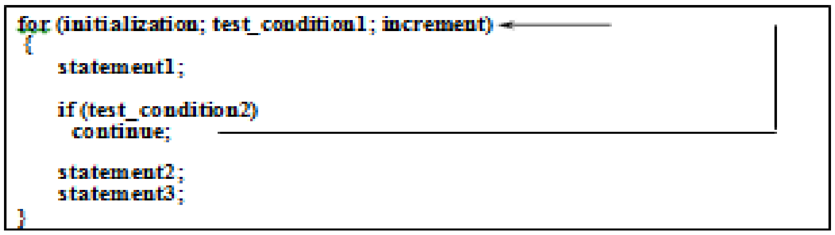
Use of the continue statement in loops:



```
while (test_condition1)
 {
      statement1;

      if (test_condition2)
         continue;

      statement2;
      statement3;
 }
```



```
do
 {
    statement1;

    if (test_condition1)
     continue;

    statement2;
    statement3;
 } while (test_condition2);
```



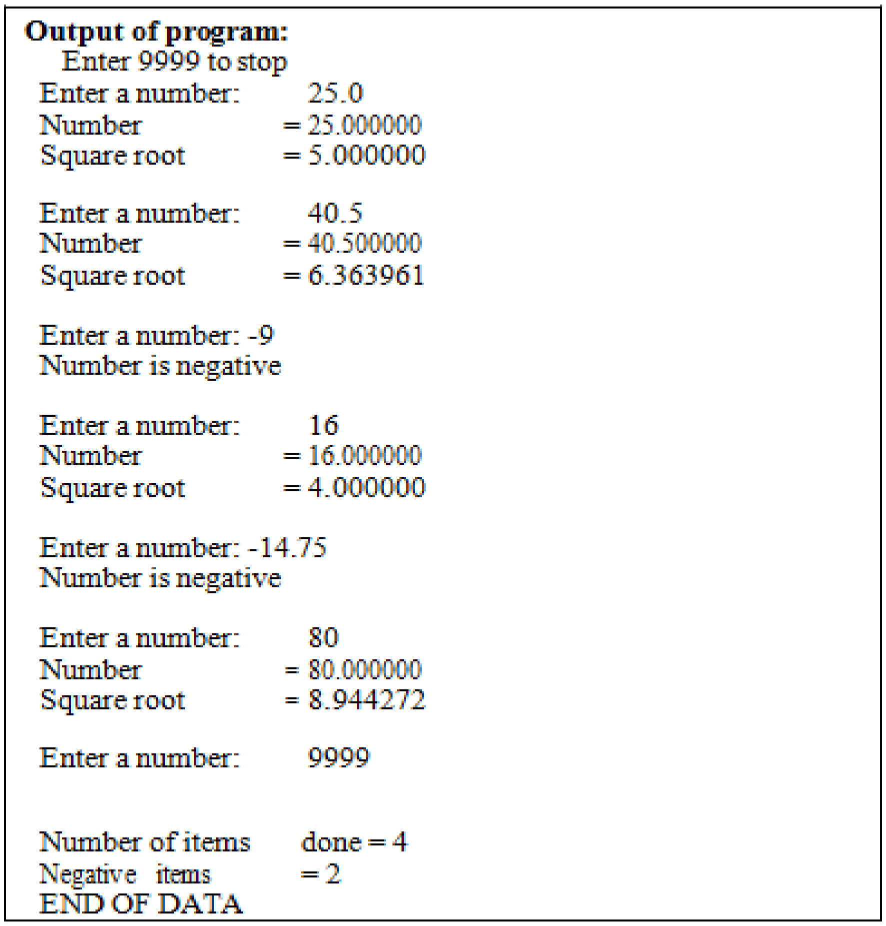
```
for (initialization; test_condition1; increment)
 {
    statement1;

    if (test_condition2)
      continue;

    statement2;
    statement3;
}
```

- In **while** and **do** loops, **continue** causes the control to go directly to the **test_ condition** and then to continue the iteration process.

- In **for** loop, the **increment** section of the loop is executed *before* the **test_ condition** is valuated.

Example: Write a C program to evaluate the square root of series of numbers and print the results. The process should stop when the number 9999 is typed. Use continue statement to skip, if *negative* number is entered.

```
/***USE OF continue STATEMENT***/
#include <stdio.h>
#include <math.h>
main()
{
        int count, negative;
        double number, sqroot;

        printf("Enter 9999 to stop\n");
        count = 0;
negative = 0;

while(count <= 100)
  {
        printf(“Enter a number:”);
        scanf(“%lf”,  &number);
        if (number == 9999)
              break;
        if (number < 0)
      {
              printf(“\n Number is negative\n\n”);
              negative++;
              continue;
       }

        sqroot = sqrt(number);
        printf(“\n Number\t\t =%lf \n Square root\t =%lf \n\n”, number, sqroot);
        count++;
  }
  printf(“\n Number of items done = %d\n”, count);
  printf(“\n\n Negative items = %d\n”, negative);
  printf(“END OF DATE\n”);
  return 0;
}
```



Output of program:

```
   Enter 9999 to stop
 Enter a number:       25.0
 Number               = 25.000000
 Square root          = 5.000000

 Enter a number:       40.5
 Number               = 40.500000
 Square root          = 6.363961

 Enter a number: -9
 Number is negative

 Enter a number:       16
 Number               = 16.000000
 Square root          = 4.000000

 Enter a number: -14.75
 Number is negative

 Enter a number:       80
 Number               = 80.000000
 Square root          = 8.944272

 Enter a number:       9999


 Number of items      done = 4
 Negative   items     = 2
 END OF DATA
```

Additional Info:

<u>Concise Test expressions:</u>

- We often use **test expressions** in the **if**, **for**, **while** and **do** statements that are *evaluated* and *<u>compared with zero</u>* for making branching decisions. Since every *integer expression* has a *true/false* value, we <u>need not make</u> <u>explicit comparisons with</u> *<u>zero</u>*.

- For instance, the *expression x* is **true** whenever *x is not zero*, and **false** when *<u>x is zero</u>*. Applying **! operator**, we can write concise test expressions <u>without using any</u> *<u>relational operators</u>*.

if (***expression*** **== 0)**

is equivalent to

if (!*expression*)

Similarly,

if (***expression*** **!= 0)**

is equivalent to

if (***expression*** **)**

Example:

if (m % 5 == 0 && n % 5 == 0)

is same as

if (!(m % 5) && !(n % 5))