ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES (AUTONOMOUS)

Programming with C - Lab

Semester - I

(Common to All Branches)

Prepared by Departments of IT & CSE

TABLE OF CONTENTS

WEEK - 1	
1	Fundamentals of Computer Hardware
2	Introduction to Programming Languages & Translators
3	DOS/UNIX Commands

Fundamentals of Computer Hardware

Introduction to Computers

The term "Computer" is derived from the word **'compute'**, which means to calculate. A computer is an electronic data processing system, which works very fast and capable of performing both arithmetic and logical functions.

Computer performs the following **3 operations** in a sequence:

1. Accepts input data
2. Stores and processes the data in rapid speeds
3. Outputs the required information in desired format.

Characteristics of a computer are:

1. responds to a specific set of instructions in a well-defined manner.
2. can execute a prerecorded list of instructions.

Advantages of computers:

1. *High speed*: Computers have the ability to perform routine tasks at a greater speed than human beings. They can perform millions of calculations in seconds.
2. *Accuracy*: Computers are used to perform tasks in a way that ensures accuracy.
3. *Storage*: Computers can store large amount of information. Any item of data or any instruction stored in the memory can be retrieved by the computer at lightning speeds.
4. *Automation*: Computers can be instructed to perform complex tasks automatically (which increases the productivity).
5. *Diligence*: Computers can perform the same task repeatedly & with the same accuracy without getting tired.
6. *Versatility*: Computers are flexible to perform both simple and complex tasks.
7. *Cost effectiveness*: Computers reduce the amount of paper work and human effort, thereby reducing costs.

Limitations of computers:

1. Computers need clear & complete instructions to perform a task accurately. If the instructions are not clear & complete, the computer will not produce the required result.

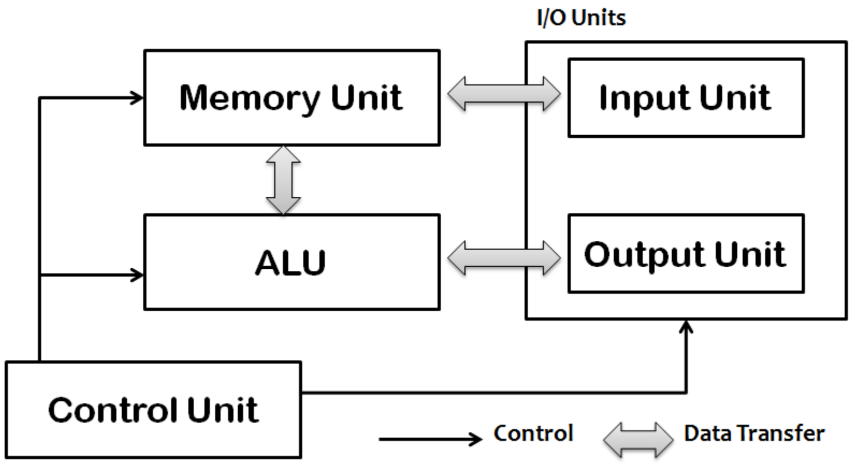
2. Computers cannot think.
3. Computers cannot learn by experience.

HARDWARE

Computer Hardware is the physical part of a computer or physical components of the computer, including the digital circuitry, as distinguished from the computer software that executes within the hardware. It refers to the objects that we can actually touch.
Ex: input and output devices, processors, circuits and the cables.

Organization of Computer

The computer consists of five functionally independent main parts:



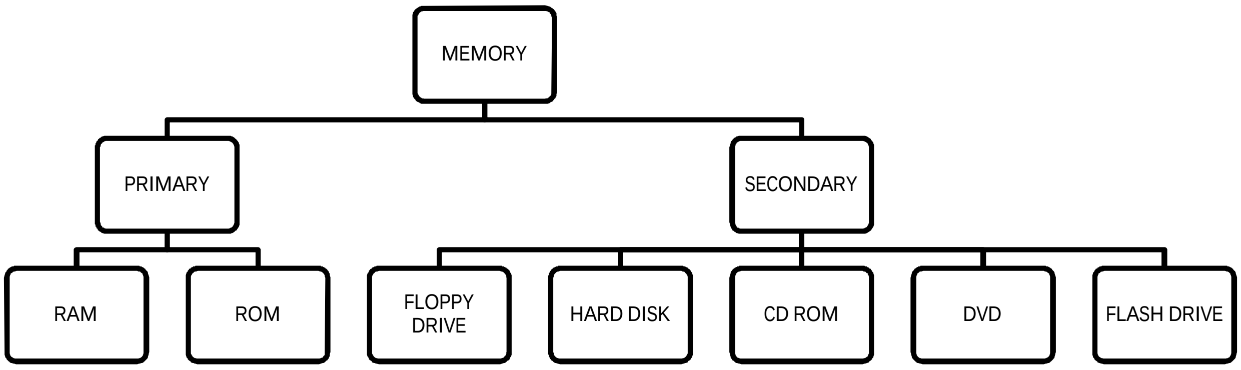
1. Input Unit
2. Memory Unit
3. A. L. Unit
4. Output Unit
5. Control Unit

<u>INPUT UNIT:</u>

- ❑ Computer accepts coded information through input units and read the data. The most well known input device is keyword.
- ❑ Whenever a key is pressed the corresponding letter or digit is automatically translated into corresponding binary code and transmitted over a cable to the memory or the processor.
- ❑ Much other type of input devices includes joystick and mouse.

<u>MEMORY UNIT:</u>

- ❑ The main function of the memory is to store program and data.
- ❑ There are two classes of storage called primary and secondary memory.



RAM (Random Access Memory): It is a temporary storage and data will be erased when the system is turned off.
ROM (Read Only Memory): It is a permanent memory and data will not be erased when the system is turned off.

DATA STORAGE

- 4bits = 1 Nibble
- 8bits = 1 byte
- 1024 bytes = 1k or 1kb (kilobyte)
- 1024KB = 1MB (mega byte)
- 1024MB = 1GB (Gega byte)
- 1024GB = 1TBC Terabytes

Although primary storage is essential but it tends to be expensive. Thus additional, cheaper secondary storage is used. Large amounts of data and programs are stored in secondary storage.

A.L.U: Most computer operations are executed in A.L.U. of the processor. A.L.U. stands for Arithmetic & Logical Unit. Consider a typical example: suppose two numbers located in the memory are two to be added, they are brought into the processor and the actual addition is carried out of the A.L.U. The sum may be stored in the memory or retain in the processor.

OUTPUT UNIT: The output unit is the counter part of the input unit. Its function is to the outside world. The most typical example of such device is printer, monitor and LCD.

CONTROL UNIT: CU controls the overall operations of the computer i.e. it checks the sequence of execution of instructions, and, controls and coordinates the overall functioning of the units of computer. The memory unit, arithmetic & logical unit, input and output unit's store and process information and perform input and output operations. Control unit must coordinate the operations of these units.

SOFTWARE

Software is a program or set of instructions that causes the Hardware to function in a desired way.

Practical computer systems divide software into three major classes:

- **System software** helps run the computer hardware and computer system. It includes operating systems, device drivers, diagnostic tools, servers, windowing systems, utilities and more.
  Example: Windows OS, Unix OS, etc
- **Programming software** usually provides tools to assist a programmer in writing computer programs and software using different programming languages in a more convenient way. The tools include text editors, compilers, interpreters, linkers, debuggers, and so on.
  Example: TurboC2, NetBeans IDE, etc
- **Application software** allows end users to accomplish one or more specific (non-computer related) tasks. Typical applications include industrial automation, business software, educational software, medical software, databases, and computer games. It is used to automate all sorts of functions.
  Example: EzSchool, Word processing tool, Games, etc

OPERATING SYSTEM

An operating system acts as an interface between user and computer hardware. It provides a user-friendly environment in which a user may easily develop and execute programs in more flexible manner.

The goals of Operating System:

1. **Convenience:** Operating System makes a computer more convenient to use.
2. **Efficiency:** Operating System allows the computer system resources to be used in an efficient manner.

A computer system has some resources which may be utilized to solve a problem. They are Memory, Processor(s), I/O, File System, etc. The OS manages these resources and allocates them to specific programs and users.

Application Areas of computers:

Computers have proliferated into various areas of our lives. These are being used in large number of areas to perform a variety of tasks. Some of the application areas of the computer are listed below:

- Education
- Entertainment
- Sports
- Advertising
- Medicine

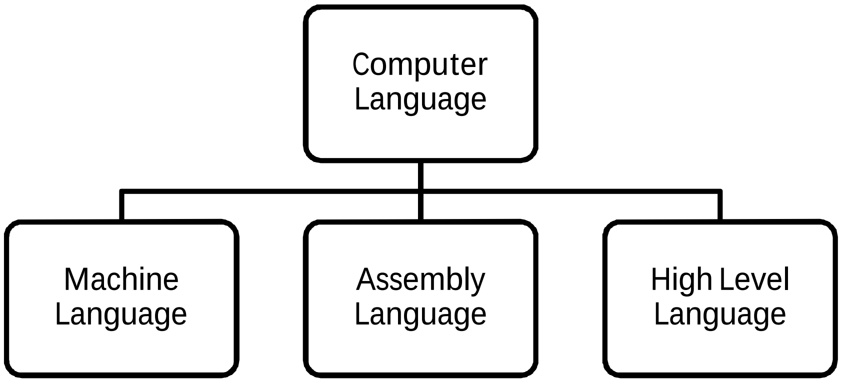
- ❑ Science & Engineering
- ❑ Government
- ❑ Home

Computers have also proliferated into areas like banks, investments, stock trading, accounting, ticket reservation, military operations, meteorological predictions, social networking, business organizations, police department, video conferencing, tele-presence, book publishing, web newspapers, and information sharing.

Introduction to Programming Languages & Translators

- **Language** is way of communication between two persons.
- **Computer Languages** are communication between computer and person.
- Computers can understand only machine instructions. Instructions are to be given in machine understandable language.
- A ***programming language*** is defined by a set of rules. It is a *formal constructed language*, designed to communication instructions to a computer. Programming languages can be used to create programs to control the behavior of the machine.
- A ***program*** is a list of instructions or statements for directing the computer to perform a required data-processing task.

TYPES



Machine language:

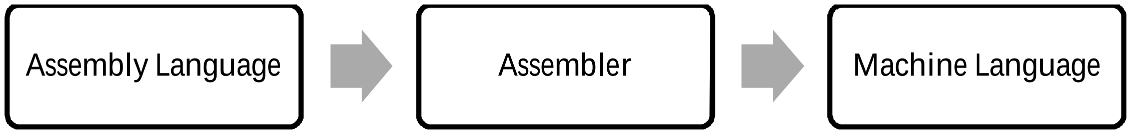
- ➢ At the lowest level computer understands only 0 and 1.
- ➢ Programs expressed in terms of binary language are called machine language and is the only one language computer can understand.
- ➢ A computer's programming language consists of strings of binary numbers (0's and 1's).
- ➢ A machine language programmer
  - o has to know the binary code for each operation to be carried out,
  - o must also be familiar with the internal organization of the computer,

- must also keep track of all the addresses of main memory locations that are referred to in the program.

- The machine language format is slow and tedious as users could not remember these binary instructions.

Assembly language and Assembler:

- A low level first generation computer language, popular during early 1960s, which uses abbreviations or mnemonic codes (mnemonic means mind full) for operation codes and symbolic addresses. This symbolic instruction language is called Assembly language.
- One of the *first steps* in improving the program preparation was to substitute mnemonics for operation codes. The mnemonics are different among makes and models of computer.
- *Second step* was symbolic addressing to express an address in terms of symbols convenient to the programmer.
- *Another improvement* was the programmer turned the work of assigning and keeping track of instruction addresses over to the computer.
- The mnemonics are converted into binaries with the help of a translator known as Assembler.



- The program written using mnemonics is called Source Program or assembly language program, the binary form of the source program equivalent is called Object Program.
- Assembler is used to convert assembly language into the machine language.
- Assembly language programs are commonly used to write programs for electronic controls using microprocessors e.g., compilers, operating systems, animation in computer graphics and so on.
- Assembly language is relatively easy for the human beings compared to machine language. Programs writing are faster compared to machine language.
- Assembly language programmer should know details of the architecture of the machine. Assembly language programs are not portable.

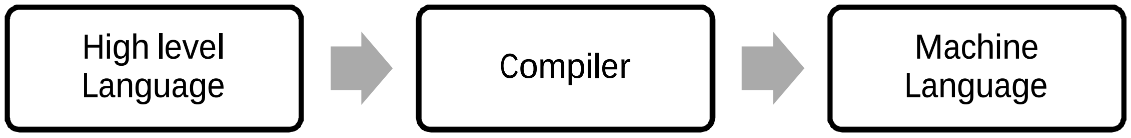
Higher level languages and compiler:

- Instructions which are written using English language with symbols and digits are called high level languages and is closer to our natural language.

- The commonly used high level languages are FORTRAN, BASIC, COBOL, PASCAL, PROLOG, C, C++, JAVA etc.
- The complete instruction set written in one of these languages is called a high level language program or computer program or source program.
- In order to execute the instructions, the source program is translated into binary form by a compiler or interpreter.
- It is also necessary to create an executable program to execute the instructions given in a source program by linking the input and output devices with your program.
- A linker (another program) is used to link library routing and generate an executable program from an object program.

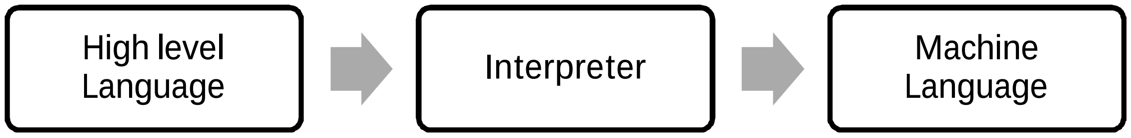
Compiler:

Compiler is a translator that converts the program instructions from human understandable form (high level language) to machine understandable form (machine language) and the translated program instruction is called object code. Every programming language requires its own compiler to translate the program. A compiler is also used to translate source program into an object program. Compiler converts source program into object program in terms of stages called passes. Normally, most of the compilers use two passes to convert source program into the machine language program.



Interpreter:

An interpreter is a program which takes the source program line by line and converts into machine code but execute each line by line as it is entered. The translation of the source program takes place for every run and is slower than the compiled code. An interpreter must accompany the object code to run a program. Programming languages BASIC, LISP, JAVA use interpreters.



DOS/UNIX Commands

DOS Commands

MS-DOS

It was developed as early as 1980 by Bill Gates at the age of 19. DOS is a single user and single task operating system. It is a character user interface operated with keyboard only. It is a collection of programs & other files. It is designed to provide a method of organizing and using the information stored on disks, application programs, system programs and the computer itself.

Files and File names: A file is a collection of related information. The files should have suitable names for their identification in later use.

Rules for naming the files: <FILENAME>.<EXTENSION>

1. File names should be of one to eight characters in length with an option of one to three character extension
2. File names can include any one of the following characters:
   A to Z (or a to z) 0 to 9, $, &, #, @. %, ( ), { }, _
3. The characters which are not allowed are: :, ; + / \ * as these have special meaning.
5. (.) is used to separate first part of file name from extension. (letter.txt etc.,)
6. When a file name includes an extension, it should be referred along with its extension and not only with the first part

Directories: It is a collection of files, size, date and time of creation of files. A directory may contain directories also. The main directory of a drive is called Root Directory into which several directories and sub-directories can exist.

Commands

1. Command: VER
   Description: It displays the version of operating system
   Syntax: C:\> VER
   Example Output: Microsoft Windows [Version 5.1.7601]

2. Command: DATE
   Description: It displays current date and asks for new date in (mm)-(dd)-(YY) format. If no date is to be changed, the old date can be retained by pressing enter key.
   Syntax: C:\> DATE
   Example Output:
   The current date is: 10/08/2012
   Enter the new date: (mm-dd-yy)

3. Command: TIME
   Description: It displays current time and asks for new tine and if no new time is to be entered, pressing enter retains the old time.
   Syntax: C:\> TIME
   Example Output:

   ```
       The current time is: 21:38:51.06
   Enter the new time:
   ```

4. Command: TITLE
   Description: Sets the window title for the command prompt window.
   Syntax: C:\> TITLE [string]
   where "string" specifies the text to set the title.
   Example Output:

   ```
   TITLE CPNMLAB
   CPNMLAB            —  ❑   X   }
                                 }
   C:\> TITLE CPNMLAB            }  COMMAND PROMPT WINDOW
   ```

5. Command: CLS
   Description: It clears the screen
   Syntax: C:\> CLS
   Example Output: Screen gets cleared and displays C:\> at the top

6. Command: [DRIVE]:
   Description: To change the drive letter in MS-DOS, type the drive letter followed by a colon.
   Syntax: C:\> [drive]:
   Example Output:

   ```
   C:\> D: {changes the drive letter from C to D}
   D:\>
   ```

7. Command: MD
   Description: It is used to make a new directory (or sub-directory) which is subordinate to the current (or root) directory.
   Syntax: C:\> MD <directory_name>
   Example Output: C:\> MD IT_24

8. Command: CD
   Description: It is used to change from one directory to the other.
   Syntax: C:\> CD <directory_name>
   Example Output:

```
C:\> CD IT_24
C:\IT_24>
```

9. Command: CD..
   Description: Goes back one directory.
   Syntax: C:\DIRECTORY_NAME>CD..
   Example Output:
   ```
   C:\IT_24>CD..
   C:\>
   ```
10. Command: CD\
    Description: Goes to the highest level, the root of the drive.
    Syntax: C:\DIRECTORY_NAME\SUB_DIRECTORY>CD\
    Example Output:
    ```
    C:\Documents and Settings\User>CD\
    C:\>
    ```
11. Command: COPY CON
    Description: It allows the creation of a file through command prompt.
    Syntax: COPY CON <FILENAME>.<EXTENSION>
    Type copy con followed by the name of the file. After this you'll be returned to a blank line, which is the start of your file. Enter the lines you want to insert into the file and when done press Ctrl + Z to create the file. If you wish to cancel the creation of the file press Ctrl + C.
    Example Output:
    ```
        D:\IT_24> COPY CON sample.txt
        Name: Santosh
        Branch: IT
        Rank: 22000
        College: ANITS
         ^ Z [Ctrl + Z]
              1 file(s) copied.
    D:\IT_24>
    ```
12. Command: TYPE
    Description: Displays the contents of a text file or files.
    Syntax: TYPE [drive:\path\filename]
    Example Output:
    ```
    D:\IT_24> TYPE sample.txt
    Name: Santosh
    Branch: IT
    ```

Rank: 22000
College: ANITS
D:\IT_24>

13. Command: EDIT
    Description: Edit allows a user to view, create, or modify their computer files.
    Syntax: EDIT [drive:\path\filename]
    To Save the file Press Alt + F, a menu appears that contain "Save" option.
    To Exit the file Press Alt + F, a menu appears that contain "Exit" option.

14. Command: DIR
    Description: displays continuously a list of files and sub-directories in a directory, displays total number of files, directories, bytes used and remaining bytes (storage).
    Syntax: D:\> DIR [drive:]
    D:\> DIR F: {displays list of files and directories from F drive.}
    Attributes:
    - DIR /p: pauses the listing whenever the screen is full. Next screen loads when any key is pressed.
    - DIR /w: displays files of directory in 5 column format. Only filenames and extensions will be displayed.
    - DIR /d: displays files in wide format and in sorted order.
    - DIR /ON: Displays all the files in A to Z order.
    - DIR /O-N: Displays all files in reverse (Z to A) order.

    Wildcards: DIR can also be specified with wild card characters (such as *) to list files sharing a common element in the filename or extension.
    - D:\> DIR *.txt : list all files with extension .txt {text files} in drive D.
    - D:\> DIR W*.*: list all files that start with W & other extensions in drive D.

15. Command: DEL
    Description: Deletes a specified file
    Syntax: DEL {filename}
    Example Output:
    D:\IT_24> DEL sample.txt : Deletes file sample.txt from IT_24 directory
    D:\IT_24> DEL *.txt : Deletes all files with extension .txt
    D:\IT_24> DEL W*.* : Deletes all files that start with W & other extensions.

16. Command: REN
    Description: Changes the name of old file with new name.
    Syntax: REN old_file_name new_file_name

Example Output:

```
D:\IT_24> REN sample.txt Santosh24.txt
```

{Changes the file sample.txt to Santosh24.txt}

17. Command: COPY

Description: Copy files from one place to another.

Syntax: COPY [drive:\path]<source file> [drive:\path]<destination>

Example Output:

```
D:\IT_24> COPY Santosh24.txt santoshit.txt
```

{Copies the file Santosh24.txt with the name santoshit.txt on the same directory}

```
D:\IT_24> COPY santoshit.txt C:\
```

{Copies the file santoshit.txt with the same name into C directory}

```
C:\> COPY D:\IT_24\*.txt C:\
```

{Copies all files with extension .txt with the same names into C directory}

```
C:\> COPY D:\W*.* C:\
```

{Copies all files that start with W & other extensions with same names into C directory}

18. Command: COLOR

Description: Sets the default console foreground and background colors.

Syntax: COLOR [attr] where attr specifies color attribute of console output.

Color attributes are specified by TWO hex digits -- the first corresponds to the background; the second the foreground. Each digit can be any of the following values:

0 = Black	6 = Yellow	C = Light Red
1 = Blue	7 = White	D = Light Purple
2 = Green	8 = Gray	E = Light Yellow
3 = Aqua	9 = Light Blue	F = Bright White
4 = Red	A = Light Green	
5 = Purple	B = Light Aqua	

Example Output:

```
C:\> COLOR 1F
```

{Changes Background Color to blue and foreground color to Bright White}

19. Command: RD

Description: Removes or deletes a directory with a condition that directory should be empty. The directory can be removed from root directory/main directory.

Syntax: RD [option] [drive:]path

Options:

- RD /s directory_name: Removes all directories and files in the specified directory and the directory itself & asks for confirmation. Used to delete the directory tree.
- RD /q directory_name: Quiet mode, do not ask if ok to remove a directory tree with /S

Example Output:

```
        D:\ANITS> RD IT
{removes IT directory if it is empty else displays Directory is not empty}
D:\ANITS> RD /s IT
IT, Are you sure (Y/N)? y
{removes IT directory even when not empty as it removes all directories and files}
D:\> RD /s /q ANITS
{removes ANITS directory without confirmation all the files.}
```

20. Command: COMP
Description: Compares contents of two files.
Syntax: COMP filename1 filename2
Example Output:

```
        D:\IT_24> COMP Santosh24.txt santoshit.txt
        Comparing Santosh24.txt and santoshit.txt...
Files compare OK
D:\IT_24> COMP Santosh24.txt aa.txt
Comparing Santosh24.txt and aa.txt...
Files are different sizes.
```

21. Command: FC
Description: Compares two files and displays the difference between them.
Syntax: FC filename1 filename2
Example Output:

```
        D:\IT_24> FC Santosh24.txt aa.txt
        Comparing files Santosh24.txt and aa.txt
        ***** Santosh24.txt
        Name: Santosh
        Branch: IT
        ***** aa.txt
        Santosh IT
        Rs 50200
        *****
```

22. Command: MORE

    Description: Allows information to be displayed one page at a time. Displays a text file to the screen.

    Syntax: MORE filename

    DIR | MORE

    Example Output:

```
        D:\> DIR | MORE
        Volume in drive D is New Volume
        Volume Serial Number is 96B5-3300
        Directory of D:\

11-10-2012  22:14    <DIR>          ANITS
28-03-2012  22:03           15,543 Books and Authors.docx
30-09-2012  16:26    <DIR>          c material
07-10-2012  12:56              186 C.txt
-- More  --
11-10-2012  22:14    <DIR>          ANITS
28-03-2012  22:03           15,543 Books and Authors.docx
30-09-2012  16:26    <DIR>          c material
07-10-2012  12:56              186 C.txt
03-02-2012  22:49           28,497 Write a C program to print all permutations
of a given string.docx
              59 File(s)     44,477,964 bytes
              20 Dir(s)  64,920,227,840 bytes free
```

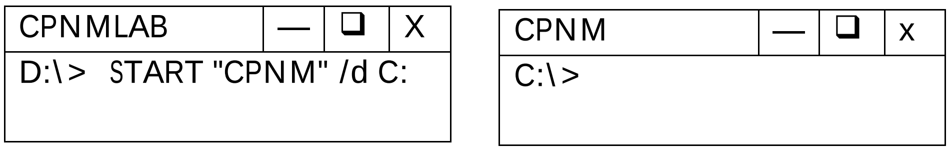
23. Command: START

    Description: Starts a separate window to run a specified program or command.

    Syntax: START [title] [/d path]

    Example Output:

```
        D:\> START "CPNM" /d C:
```



```
CPNMLAB   — ❑ X
D:\>  START "CPNM" /d C:
```

```
CPNM   — ❑ x
C:\>
```

24. Command: FIND

    Description: Searches for a text String in a file

    Syntax: FIND [OPTION] "string" file

    Example Output:

```
    D:\IT_24> FIND "e" Santosh24.txt
```

```
---------- SANTOSH24.TXT
Name: Santosh
College: ANITS
D:\IT_24> FIND /v "e" Santosh24.txt [Display lines not containing string]
---------- SANTOSH24.TXT
Branch: IT
Rank: 22000
D:\IT_24> FIND /c "e" Santosh24.txt [Display count of lines containing string]
---------- SANTOSH24.TXT: 2
D:\IT_24> FIND /n "e" Santosh24.txt [Display lines with numbers containing
string]
---------- SANTOSH24.TXT
[1]Name: Santosh
[4]College: ANITS
D:\IT_24> FIND /i "E" Santosh24.txt [Ignore case of characters]
---------- SANTOSH24.TXT
Name: Santosh
College: ANITS
```

25. Command: EXIT
    Description: Quits the CMD.EXE program (command interpreter).
    Syntax: EXIT
    Example Output:

```
D:\> EXIT {closes the command window}
```

26. Command: ATTRIB
    Description: Displays or changes file attributes.
    Syntax: ATTRIB [+R | -R] [+H | -H] [drive:][path][filename]

```
+    Sets an attribute
-    Clears  an attribute
H    Hidden File Attribute
 R   Read-Only File Attribute
```

Example Output:

```
D:\IT_24> ATTRIB +R Santosh24.txt
{Sets the read only attribute to the file and cannot be modified}
D:\IT_24> ATTRIB -R Santosh24.txt
{Clears the read only attribute to the file and can be modified}
D:\IT_24> ATTRIB +H Santosh24.txt
{Sets the hidden attribute to the file and cannot be viewed}
D:\IT_24> ATTRIB -H Santosh24.txt
```

{Clears the hidden attribute to the file and can be viewed}

27. Command: PROMPT
    Description: Changes the cmd.exe command prompt.
    Syntax: PROMPT [text] $G
    Example Output:

```
D:\> PROMPT CPNM$G
CPNM>
```

{Use the DOS commands to execute, stays in current directory with given prompt}

{to come to actual prompt, type 'prompt' and then press enter}

28. Command: PATH
    Description: Displays or sets a search path for executable files & used to provide access to files located on other directories.
    Syntax: PATH = "[[drive:]path";
    Example Output:

```
D:\IT_24> PATH = "C:\WINDOWS\system32";
```

{Sets the path for system32 directory to invoke the files required by DOS}

29. Command: SORT
    Description: Sorts input in alphanumeric order.
    Syntax:

```
SORT [/r] (enter)
        [input text {enter}]
        Ctrl + Z
        [output text {sorted order}]
```

Example Output:

```
D:\IT_24> SORT                      | D:\IT_24> SORT /r
hat                                 | hat
cat                                 | cat
mat                                 | mat
bat                                 | bat
ant                                 | ant
 ^ Z [Ctrl + Z]                     |  ^ Z [Ctrl + Z]
ant                                 | mat
bat                                 | hat
cat                                 | cat
hat                                 | bat
mat {Displays in A To Z order}      | ant {Displays in Z To A order}
```

30. Command: TREE

    Description: Displays the directories and subdirectories existing in a drive in a Tree diagram without files. Only directories with <DIR> are displayed.

    Syntax: TREE

    Example Output:

```
D:\network lab>TREE
Folder PATH listing
Volume serial number is 2475-5834
D:\NETWORK LAB
├───FTP
├───http
├───new tftp
├───remotehost
└───telnet
    └───telnet
```

UNIX Commands

UNIX is one of the most versatile and popular operating systems in the market today. It was designed and developed in 1969 to provide an environment to create programs. It became popular with its usage beginning to spread to educational institutions scientific research laboratories and industries. **Ken Thompson** and **Dennis Ritchie** created UNIX.

Basic UNIX Commands

man: The **man** command displays the online manual pages. Reads the manual page for a command.
Syntax: man command_name

ls: It is used to list all the files and directories.

ls – l: It is called as the long list. It displays or lists all the files with their preferences.

ls – a: This command is used to view all the hidden files.

mkdir: This command is used to create a directory. **Syntax: mkdir dirname**.

cd: This command is used for changing the directory

cd directoryname. This command moves to the named directory.

cd. : Changes the home directory.

cd .. : Changes the parent directory.

pwd: This command displays the path of the current directory.

cp: Copies the contents of one file to another. **Syntax: cp file1 file2**

mv: This command is used to move or rename the files. **Syntax: mv file1 file2**

rm: This command is used for removing or deleting a directory. **Syntax: rm file**

rmdir: This command is used for removing or deleting a directory.
Syntax: rmdir directory
cat: Displays a file.
Syntax: cat filename
cat: Concatenate files.
Syntax: cat file1 file2 >file3

  >: Redirects standard output to a file.
    Syntax: cat > filename

  >>: Append standard output to a file.
    Syntax: cat >> filename

  <: Redirect standard input from a file.
    Syntax: command < filename

more: Displays a file a page at a time.
Syntax: more filename

head: Displays the first few lines of a file.
Syntax: head -n/+n filename

  -n: n lines from the beginning are printed.
  +n: n-1 lines are skipped or starts from nth line.

tail: Displays the last few lines of a file.
Syntax: tail -n filename

  -n: n lines from end are displayed.

wc: Count number lines/words/characters in file.
Syntax: wc <option(s)> filename

  -w: To do a word count.
  -l: To find out how many lines the file has.

grep: Grap regular expressions and patterns. It is used to search for a particular word in a file.
Syntax: grep <option(s)> keyword filename

  -i: Ignores the case of the word.
  -v: Prints the lines that do not match.

-c: Prints only the total count of matched lines.

pipe: It is used to join two or more commands where output of command1 is the input of command2.
Syntax: command1| command2

sort: Sorts the content or data.
Syntax: sort <option(s)> filename
-n: Numeric sort.
-r: Reverse sort.
-d: Directory sort.
who: Lists users currently logged in.
Syntax: who >filename

?: Matches one character.
***:** Matches any number of records.

what is: Offers brief description of a command.

apropos: Matches command with the specified keyword in their man pages.
Syntax: apropos keyword.

ps: List current processes.

chmod: Changes the access rights for named file.
Syntax: chmod [options] filename
-u: User
-g: Group
-o: Others
-a: All
-r: Read
-w: Write
-x: Execute
-+: Add permission
--: Take away permission

df: Reports the space left on the file system.
Syntax: %df

du: The number of kilobytes used by each subdirectory.
Syntax: %du

nl: Appends line number. No number for empty files.

egrep: Useful for multiple patterns matching can be made file based by –f option.
Syntax: egrep <options> filename

-c+: One or more occurrences of character.
-c?: Zero or one occurrences of character.
-a/b: Either a or b.
-(a): Regular expression.

fgrep: Fast or fixed grep. It does not accept regular expression. In this one command is separated from other by new line.
Syntax: fgerp 'abc newline xyz'

tee: Directs the output to the standard output device.
Syntax: tee <option> filename
-a: Appends to a file.

find: Searches for a file.
Syntax: find <option> filename

-atimen: True if file was accessed n days ago
-ctimen: True if file was created n days ago.
-namepattern: True if filename matches pattern.
-print: Print names of files found.

Cmp: Purpose: Compare two files
Syntax: cmp file1 file2 [skip1 [skip2]]
ReturnType:

0-files are identical
1-files are different.
>1-an error occurred.

Description: The cmp utility compares two files of any type and writes the results to standard output. By default cmp is silent if files are same; if they differ, the byte and line number at which the first difference occurred is reported. Bytes and lines are numbered beginning with one.
Options:

-l prints the byte number and differing byte values for each difference.
-s prints nothing for differing files. Return exit status only.

The optional arguments skip1 and skip2 are the byte offsets from the beginning of file1 and file2, respectively, where the comparison will begin.
Dept. of IT, ANITS Page 17
Example: $ cmp group1 group2 group1 group2 differ:cahr47,line3

Diff: Purpose: Find differences between two files.
Syntax: diff [options] from-file to-file.
Return Type:

-0 no difference

-1 some difference were found
-2 trouble.

Description: -diff compares contents of the two files from-file and to-file.
-A file name of –stands for text read from standard input.
-As special case diff –compares copy of standard input to itself.
-If from file is a directory and to-file is not, diff compares the file in from-File whose name is that of to-file and vice-versa. The non directory file must not be.
-If both from-file and to-file are directories, diff compares corresponding Files in both directories, in alphabetical order.

Options: a great all files as text and compare then line by line, even if they do not seem to be next. Ignore changes in amount of white space.

WEEK - 2	
1	Algorithms & Flowcharts
2	Introduction to C, History, Steps of Learning C
3	C – Tokens, Data Types - Format Specifiers, I/O Statements
4	Structure of C Program, Sample C Programs

Algorithms & Flowcharts

Algorithms

Algorithm is a method of representing the step-by-step logical procedure for solving a problem.
A program written in a non-computer language is called an algorithm. It is a step-by-step method of performing any task.
These are one of the most basic tools that are used to develop the program solving logic. They can have steps that repeat or require decisions until the task is completed.
PROPERTIES
An algorithm must possess the following properties:

1) **Fitness:** An algorithm must terminate in a finite number of steps.
2) **Definite:** By definite we mean that each step of algorithm must be precisely defined such that there is no ambiguity or contradiction.
3) **Effectiveness:** Each step must be effective, easily converted into program statement and can be performed exactly in a finite amount of time.
4) **Generality:** The algorithm must be complete so that it will work successfully in solving all the problems of particular type for which it is defined.
5) **Input/Output:** Each algorithm must take zero, one or more quantities an input data and produce one or more output values.

While writing an algorithm we can concentrate only on the logic and not on language syntax and once the algorithm is written we can code the algorithm in computer language.

Example 1: Write an algorithm to find the sum and average of two numbers

```
Step 1. Read the numbers a, b
Step 2. Compute the sum of a & b
Step 3. Store the result in variable s
Step 4. Divide the sum s by 2
Step 5. Store the result in variable avg
Step 6. Print the value of s and avg
Step 7. End of program.
```

Example 2: Find Area, Diameter and Circumference of a Circle.

```
Step 1: Start
Step 2: Initialize PI to 0
Step 3: Read radius of the circle
Step 4: Calculate the product of radius with itself and PI value
Step 5: Store the result in variable area
Step 6: Calculate the product of radius with 2
Step 7: Store the result in variable diameter
Step 8: Calculate the product of PI and diameter value
Step 9: Store the result in variable circumference
Step 10: Print the value of area, diameter and circumference
Step 11: Stop
```

Example3: Find Area and Circumference of a Rectangle.

```
Step 1: Start
Step 2: Read the values of length and breadth
Step 3: Calculate the product of length and breadth
Step 4: Store the result in variable area
Step 5: Calculate the sum of length and breadth
Step 6: Store the result in variable temp
Step 7: Calculate the product of 2 and temp
Step 8: Store the result in variable circumference
Step 9: Print the value of area and circumference
Step 10: Stop
```

Flowcharts

INTRODUCTION

The flowchart is a mean of visually presenting the flow of data through an information processing systems, the operations performed within system and the sequence in which they are performed.

A flowchart is a pictorial representation of an algorithm in which the steps are drawn, in the form of different shapes of boxes and the logical flow indicated by inter connecting arrows.

Meaning of a Flowchart

A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

ADVANTAGES:

Reasons for using flowcharts as a problem-solving tool are:

- Makes logic clear
- Communication: flowcharts are a good way of communicating the logic of a system to all concerned.
- Using in coding, flowcharts act as a guide during the system's analysis and program development phase.

- Effective analysis: problem can be analyzed in more effective way.
- Efficient program maintenance: The maintenance of a program becomes easy with the help of flowchart.
- Proper documentation: flowcharts serve as a good program documentation.

DISADVANTAGES:

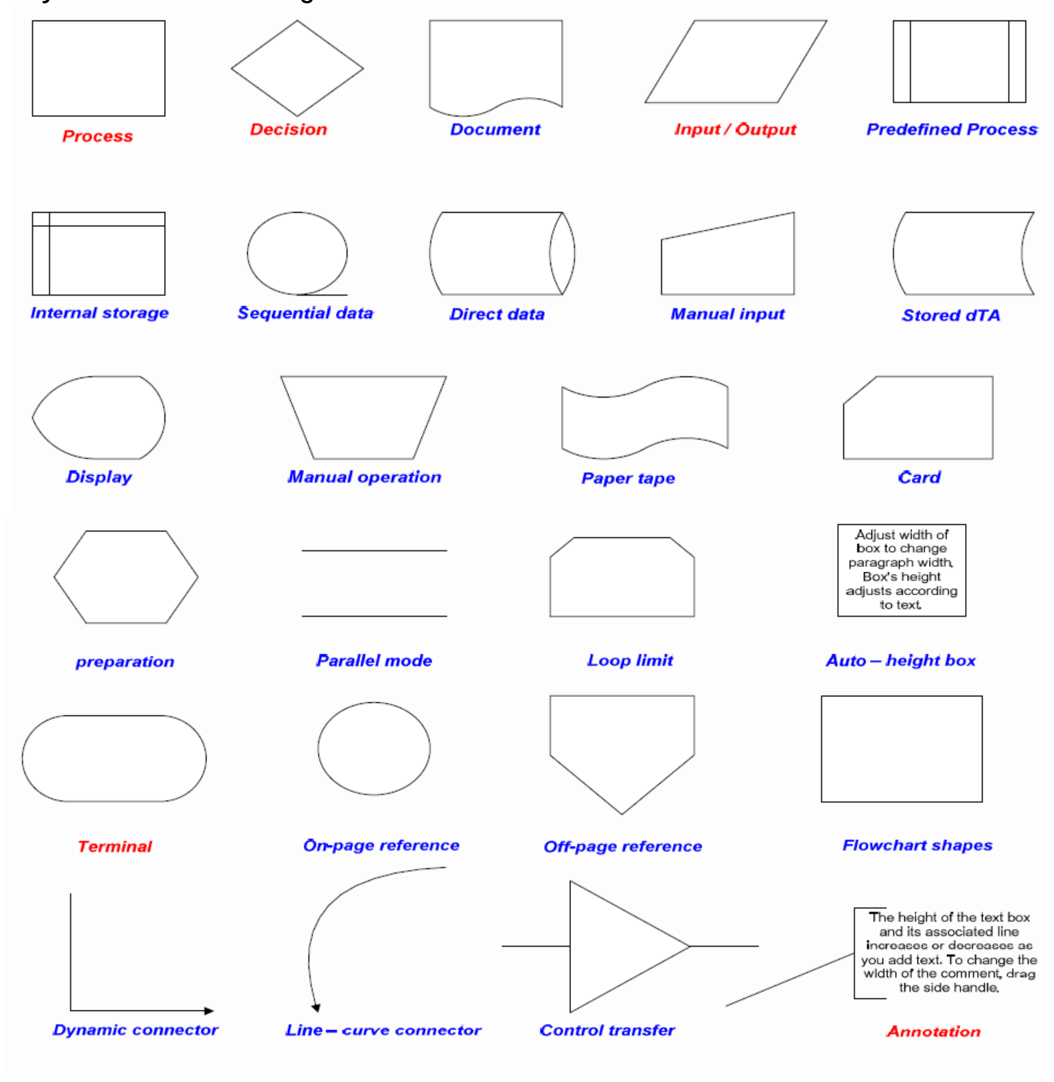
a) Complex logic: Sometimes, the program logic is quite complicated. In this case, flowchart becomes complex and clumsy.

b) Alterations and modifications: If alterations are required the flowchart may require re-drawing completely.

c) The essentials of what is done can easily be lost in the technical details of how it is done.

SYMBOLS:

On flowcharts different geometric shapes are used called flowchart symbols. Some standard symbols for drawing flowcharts are:



Basic Flowchart Shapes

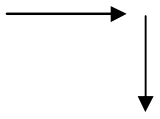
The basic 5 flowchart symbols are

1) Terminal block
2) Input / Output
3) Process block
4) Decision block
5) Flow lines.

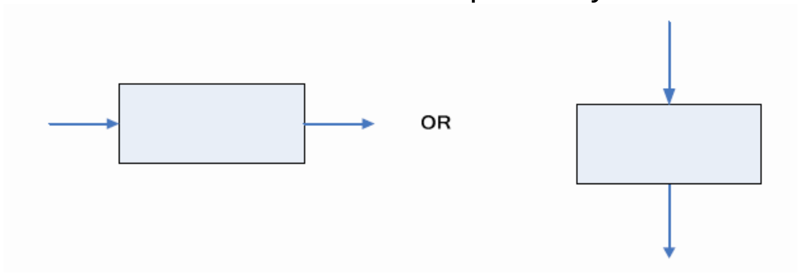
GUIDELINES:

The following are some guidelines in flowcharting:

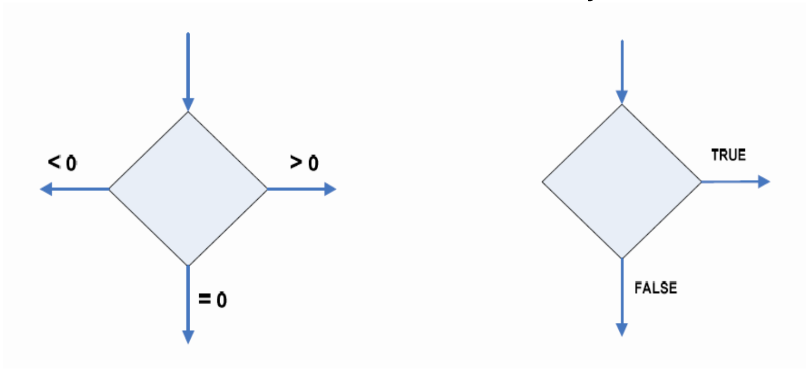
a) The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
b) The usual direction of the flow of a procedure or a system is from left to right or top to bottom.



c) Only one flow line should come out or from a process symbol.

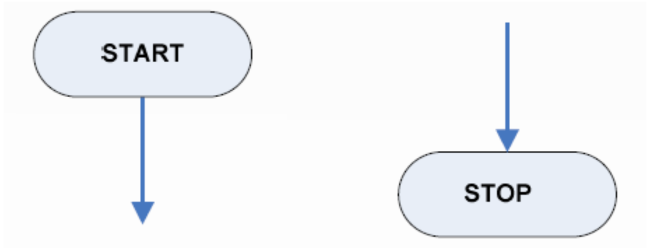


d) Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.

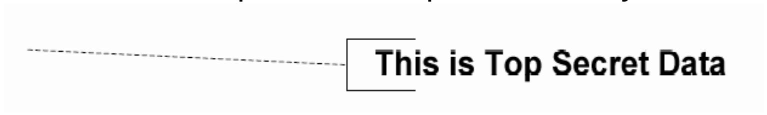


Three-way branch decision ***Two-way branch decision***

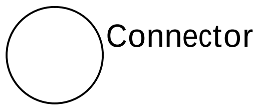
e) Only one flow line is used in conjunction with terminal symbol.



f) Write within standard symbols briefly. As necessary, you can use the annotation symbol to describe data or computational steps more clearly.



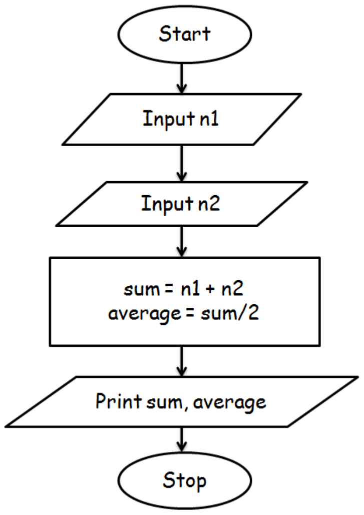
g) If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.



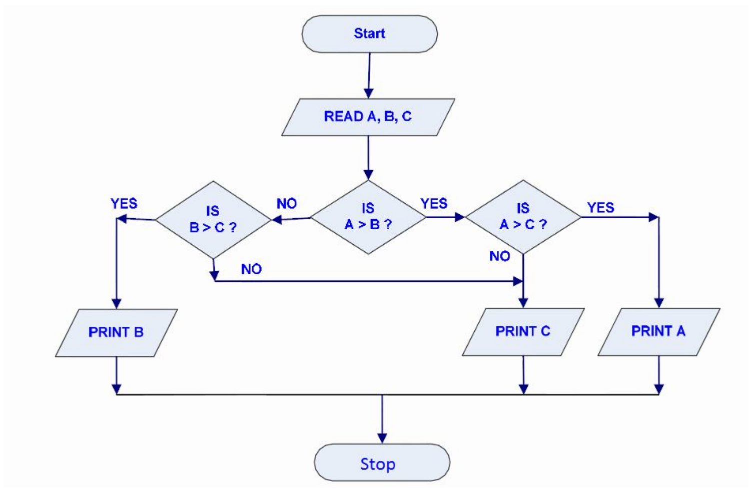
h) Ensure that the flowchart has a logical start and finish.

i) It is useful to test the validity of the flowchart by passing through it with a simple test data.

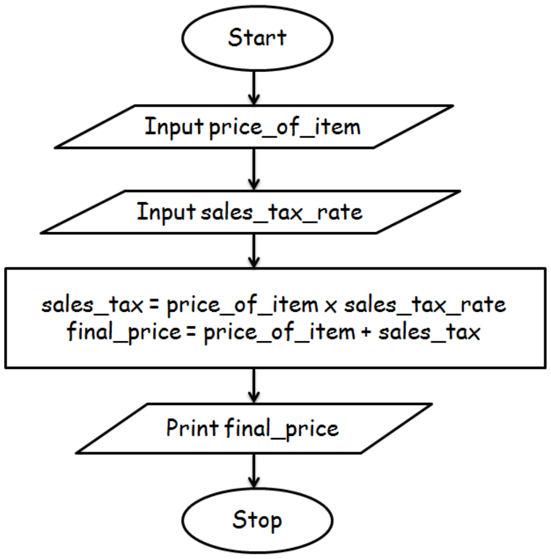
Example 1. Draw a flowchart to find the sum and average of two numbers.



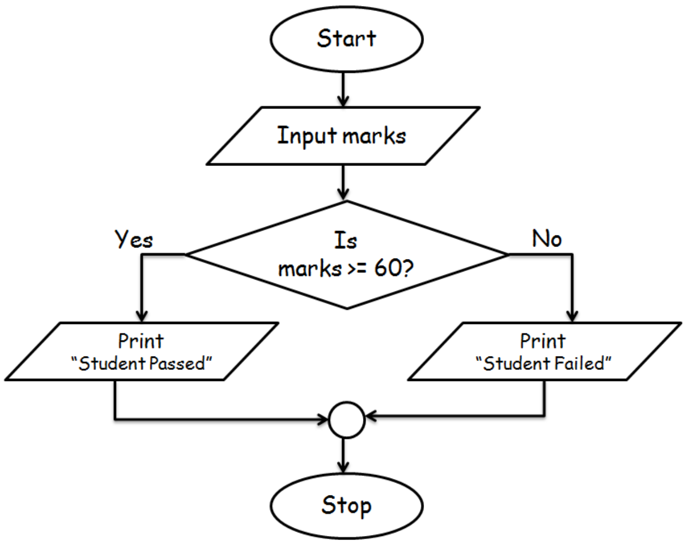
Example 2. Draw a flowchart to find the largest of three numbers.



Example 3: Draw a flowchart to compute the final price of an item after figuring in sales tax.


Start
Input price_of_item
Input sales_tax_rate
sales_tax = price_of_item x sales_tax_rate
final_price = price_of_item + sales_tax
Print final_price
Stop

Example 4: Write a pseudo code to check whether a student is passed or not.

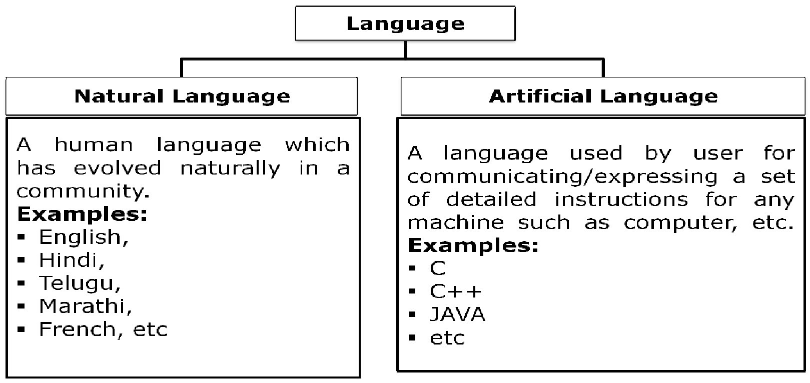


Differences between Algorithm and Flowchart

Algorithm	**Flowchart**
Step by step procedure for solving a problem	Pictorial representation of an algorithm
Written in English-like language with words	Steps of algorithm are drawn in the form of shapes of boxes and logical flow with arrows.
Explains how a certain process is followed or a problem solved	May not have detailed instructions about how the tasks are done.

Introduction to C Language, History, Steps in Learning

Language is the expression of thought in a specified way. It is a body of words, and set of methods of combining them (called grammar), understood by a community and used as a form of communication.


Language
Natural Language
A human language which has evolved naturally in a community.
Examples:
English,
Hindi,
Telugu,
Marathi,
French, etc
Artificial Language
A language used by user for communicating/expressing a set of detailed instructions for any machine such as computer, etc.
Examples:
C
C++
JAVA
etc

INTRODUCTION

C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 at Bell Labs. C is a robust language whose rich set of built-in functions and operators can be used to write any complex program. C is one of the most widely used programming languages of all time.

Origin of C

Year	Language	Developed By	Remarks
1960	ALGOL60	International Committee	too general, too abstract, not a specific language
1963	CPL(Combined Programming Language)	Cambridge University	too difficult to learn and hard to implement
1967	BCPL(Basic Combined Programming Language)	Martin Richard, Cambridge University	could deal only special problem
1970	B	Ken Thompson, AT & T Bell Laboratories	could deal only special problem, did not support various data types
1972	C	Dennis Ritchie, AT & T Bell Laboratories	programming efficiency & relatively good machine efficiency & support various data types.

C was the offshoot of the earlier languages ‘BCPL’ and ‘B’. It supports high level features with commands like if, else, goto etc making it convenient to use. It also supports low-level features like manipulation of bits and addressing memory location directly by using pointers. C may be called as **middle level programming language**.

C compiler combines the capabilities of an assembly language with features of high-level language and therefore it is well suited for writing both system software and business packages.

Application Areas of C

- Initially C was used to design the system software like 90% of UNIX operating system is written in C language.
- for designing application software
- In writing Device Drivers.
- In INTERNET protocols.
- In developing translators.
- In print spoolers and other utilities.

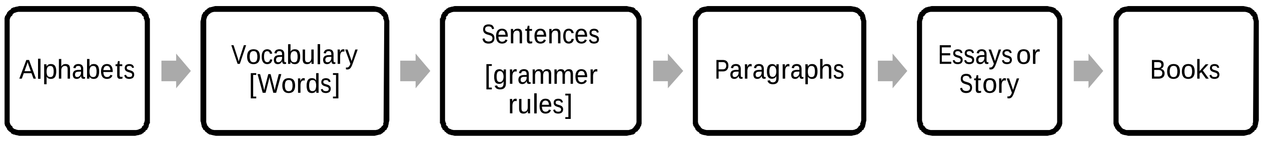
Getting Started With C

Communicating with a computer involves speaking the language the computer understands, which immediately rules out English as the language of communication with computer. However, there is close analogy between learning English language and learning C language.

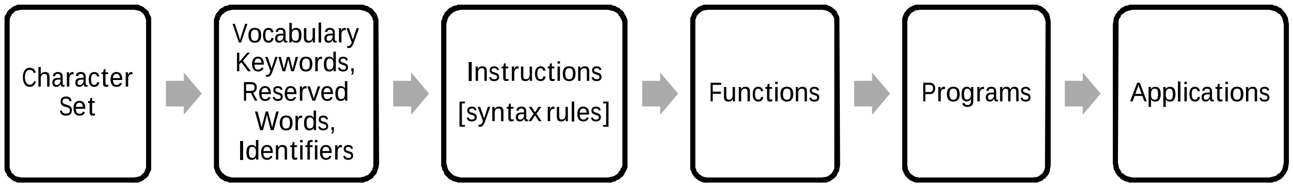
The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs.

Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables, keywords, and reserved words are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form functions or program.

Steps in learning English Language:



Steps in learning C language:


Character Set
Vocabulary Keywords, Reserved Words, Identifiers
Instructions [syntax rules]
Functions
Programs
Applications

Character Set

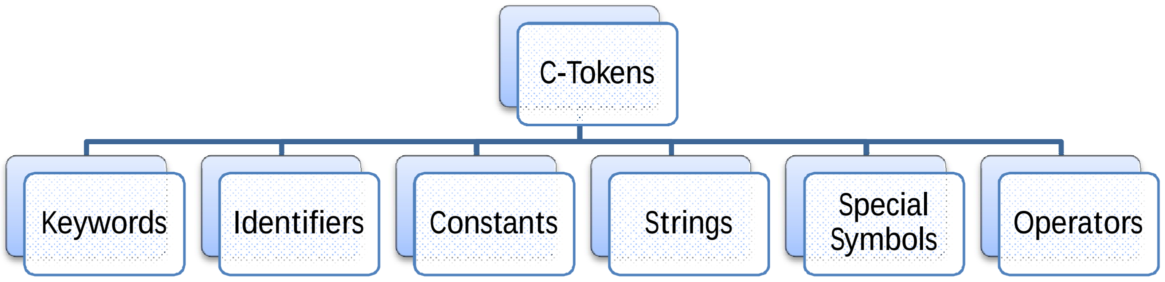
Any symbol is referred as character in C language. A character denotes any alphabets (A,B, … Z or a,b, … z), digits (0,1,2,3,4,5,6,7,8,9) or special symbols (!@#~`% ^ &*(){}-=+_{}[]|\”/’<>:;) used to represent information. This character set is used to form vocabulary of C (constants, variables and keywords).

ASCII Table								
ASCII	**Symbol**	**Description**	**ASCII**	**Symbol**	**Description**	**ASCII**	**Symbol**	**Description**
32		Space	64	@	at symbol	96	`	grave accent
33	!	exclamation	65	A		97	a	
34	"	double quotes	66	B		98	b	
35	#	number sign	67	C		99	c	
36	$	dollar sign	68	D		100	d	
37	%	percent sign	69	E		101	e	
38	&	ampersand	70	F		102	f	
39	'	single quote	71	G		103	g	
40	(	opening parenthesis	72	H		104	h	
41	)	closing parenthesis	73	I		105	i	
42	*	asterisk	74	J		106	j	
43	+	plus sign	75	K		107	k	
44	,	comma	76	L		108	l	
45	-	Minus/hyphen	77	M	UPPER CASE ALPHABETS	109	m	LOWER CASE ALPHABETS
46	.	period	78	N		110	n	
47	/	slash	79	O		111	o	
48	0	zero	80	P		112	p	
49	1	one	81	Q		113	q	
50	2	two	82	R		114	r	
51	3	three	83	S		115	s	
52	4	four	84	T		116	t	
53	5	five	85	U		117	u	
54	6	six	86	V		118	v	
55	7	seven	87	W		119	w	
56	8	eight	88	X		120	x	
57	9	nine	89	Y		121	y	
58	:	colon	90	Z		122	z	
59	;	semicolon	91	[	opening bracket	123	{	opening brace
60	<	less than sign	92	\	backslash	124	|	vertical bar
61	=	equal sign	93	]	closing bracket	125	}	closing brace
62	>	greater than	94	^	caret	126	~	equivalency-tilde
63	?	Question mark	95	_	underscore			

C TOKENS

In a C Program, the smallest individual units are known as C -Tokens. Programs are written using these tokens and syntax of language. There are totally six tokens. They are:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols



Keywords:

Keywords are those words, whose meaning is already known to the C compiler i.e. they are predefined words. The keywords cannot be used as variable names. There are only 32 keywords available in C. A keyword can't be used as a variable name because if we do so, we are trying to assign a new meaning to the keyword.

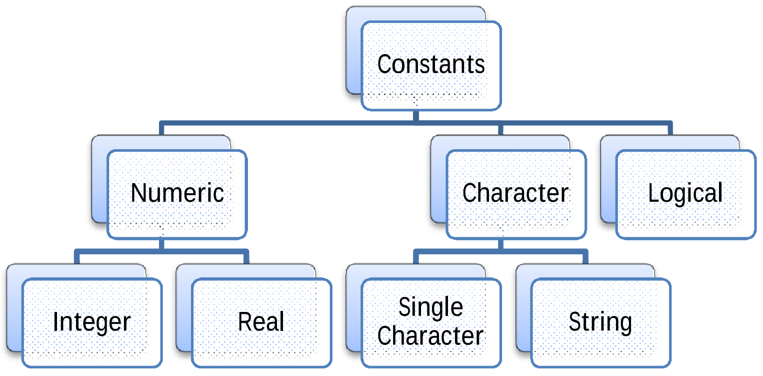
int	struct	for	static
float	union	switch	register
char	typedef	case	extern
double	enum	default	void
signed	if	break	volatile
unsigned	else	continue	sizeof
long	while	goto	const
short	do	auto	return

Identifiers:

Identifiers refer to the names of variable, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers.

Constants:

Constants in 'C' refer to fixed values that do no change during the execution of a program. Constant is a memory location in which a value can be stored and this cannot be altered during the execution of program. 'C' supports several types of constants. They are illustrated below.



Integer Constants:

An integer constant refers to a sequence of digits. There are three types of integers, namely binary, decimal, octal and hexadecimal.

Binary Numbers: A Binary Number is made up of only **0**s and **1**s.

Example: 110100

Representation of a Binary Number

MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional - or + sign. Spaces, commas and non-digit characters are not permitted between digits.

Valid Examples	Invalid Examples
123	15 750
-321	20,000
0	2.5
654321	$1000
+78	

Conversion of Binary to Decimal

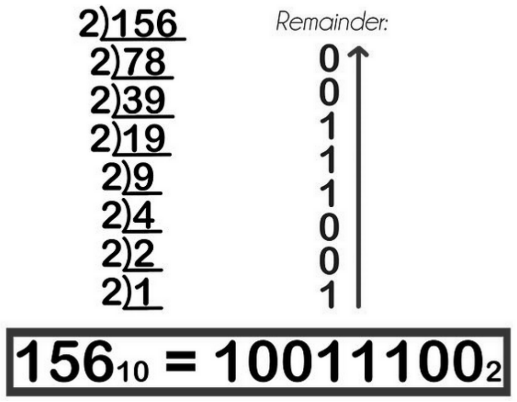
1011

You would this interpret in decimal as:

	Eights (8)	Fours (4)	Twos (2)	Ones (1)
	1×2^3	plus 0×2^2	plus 1×2^1	plus 1×2^0
=	1 x 8	plus 0 x 4	plus 1 x 2	plus 1 x 1
=	8	plus 0	plus 2	plus 1

The total is: 11 (in decimal) which equal 8 + 0 + 2 + 1
In other words,
$1011_2 = 11_{10}$

Conversion of Decimal to Binary



Octal Integers constant consists of any combination of digits from 0 through 7 with a 0 at the beginning. Octal is fancy for Base Eight meaning eight symbols are used to represent all the quantities. They are 0, 1, 2, 3, 4, 5, 6, and 7. Some examples of octal integers are 026, 0, 0347, and 0676

Octal	0	1	2	3	4	5	6	7	10	11	12…	17	20…	30…	77	100
Decimal	0	1	2	3	4	5	6	7	8	9	10…	15	16…	24…	63	64

Octal to Decimal Conversion
Just like how we used powers of ten in decimal and powers of two in binary, to determine the value of a number we will use powers of 8 since this is Base Eight. Consider the number 3623 in base eight.

8^3	**8^2**	**8^1**	**8^0**
3	6	2	3
1536+384+16+3			
1939			

Hexadecimal integer constant is preceded by OX or Ox, they may contain alphabets from A to F or a to f. The alphabets A to F refer to 10 to 15 in decimal digits. The hexadecimal system is Base Sixteen.
Examples of valid hexadecimal integers are 0X2, 0x8C, 0Xbcd, and 0x

Hexadecimal	9	A	B	C	D	E	F	10	11…	19	1A	1B	1C…	9F	A0
Decimal	9	10	11	12	13	14	15	16	17	25	26	27	28	159	160

Digits are explained as powers of 16. Consider the hexadecimal number 2DB7.

16^3	**16^2**	**16^1**	**16^0**
2	D	B	7
8192+3328+176+7			
11703			

***Real Constants*:**
Real Constants consists of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously. These quantities are represented by numbers containing fractional parts like 26.082.
Examples of real constants are 0.0026, -0.97, 435.29, +487.0

These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point or digits after the decimal point. That is, 215.; .95; -.71; +.5 are all valid real numbers.

A real number may also be expressed in exponential (or scientific) notation. For example, 215.65 may be written as 2.1565e2 in exponential notation, e2 means multiply by 10^2. The general form is:

$$mantissa\ \boldsymbol{e}\ exponent$$

The $mantissa$ is either a real number expressed in decimal notation or an integer. The $exponent$ is an integer number with an optional plus and minus sign. The letter $\boldsymbol{e}$ separating the mantissa and the exponent can be written in either lowercase or uppercase. The exponent causes the decimal point to "float", this notation is said to represent a real number in floating point form.
Examples of legal floating point constants are: 0.65e4, 12e-2, 1.5e+5, 3.18E3, -1.2E-1

White spaces are not allowed. Exponential notation is useful for representing numbers that are either very large or very small is magnitude. E.g. 7500000000 --> 7.5e9 or 75e8.

Single Character Constants
A singe character constant contains a single character enclosed within a pair of single quote marks.
Examples of single character constants are: '5', 'x', ' ', ';'.
The character constant '5' is not same as the number 5.

String Constants

A string constant is a sequence of character enclosed in double quotes. The characters may be letters, numbers, special characters and blank space.
Examples are "Hello!", "1987", "WELL DONE", "5+3", "X"

***Logical Constant*:** A logical constant can take either a true or a false as value.
In C,

- Zero (0) represents false value.
- Any non-zero value such as 1 represents true value.

Backslash Character Constants [Escape Sequences]

Backslash character constants are special characters used in output functions. Although they contain two characters they represent only one character. Given below is the table of escape sequence and their meanings.

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

Variables:

In C, a quantity, which may vary during the program execution, is called a variable. Variable names are the names given to the memory locations of a computer where the variable value is stored.
A variable is a name that is used to store data value and is allowed to vary the value during the program execution. A variable is able to hold different values during execution of a program, where as a constant is restricted to just one value. Variable name

can be chosen by the programmer in a meaningful way so as to reflect its nature in the program.

Length of C variable name can be upto 8 characters and some translators permit even a higher length (32). It may take different values at different times during program execution.

Variable names may consist of letters, digits, and underscore subjected to following conditions:

- They must begin with a letter or underscore (_), followed by any number of letters, digits, or underscores.
- Uppercase is different from lowercase, so the names total, Total, and TOTAL specify three different variables.
- The variable name shouldn't be a keyword.
- Blank spaces are not allowed.
- The length should be normally 8 characters (since only the first 8 characters treated as significant by many compilers).
- The variable names are case sensitive.
- The variables are defined at the beginning of the block.

For example, in the equation $2x + 3y = 10$; since x and y can change, they are $variables$, whereas 2,3 and 10 cannot change, hence they are $constants$. The total equation is known as $expression$.

Valid variable names: Alpha, X, fyear_9899, matrix, income

DATATYPES

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement.

Data type is the instruction to the compiler telling about what type of value will be stored in a memory location and also to specify the amount of memory required for location.

C language data types can be broadly classified as

- Primary data types
- Derived data types
- User-defined data types

Primary Data Types

Integer Type:

Integers are whole numbers with a machine dependent range of values. Generally, integers occupy one word of storage and since word sizes of machines vary (typically, 16 or 32 bits), the size of an integer that can be stored depends on the computer.
C has 3 classes of integer storage namely $short\ int, int\ and\ long\ int$. All of these data types have signed and unsigned forms. Signed numbers are positive or negative and one bit is used for sign and rest of the bits for the magnitude of the number. Unsigned numbers are always positive and consume all the bits for the magnitude of the number.

A $short\ int$ represents small integer values and requires half the space than normal integer values. The $long\ and\ unsigned\ long\ int$ are used to declare a longer range of values.

Type	Size (in Bytes)	Size (in Bits)	Range
$int\ or\ signed\ int$	2	16	$-32768\ to\ 32767$
$unsigned\ int$	2	16	$0\ to\ 65535$
$short\ int\ or\ signed\ short\ int$	1	8	$-128\ to\ 127$
$unsigned\ short\ int$	1	8	$0\ to\ 255$
$long\ int\ or\ signed\ long\ int$	4	32	$-2147483648\ to\ 2147483647$
$unsigned\ long\ int$	4	32	$0\ to\ 4294967295$

To find the range of these signed data type we use the formula, $-2^{n-1}\ to\ +2^{n-1}-1$
To find the range of these unsigned data type we use the formula, $0\ to\ 2^{n}-1$

Character Data Type

A single character can be defined as a character ($char$) type data. Characters are usually stored in 8 bits (1 byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to char. Unsigned characters have values between 0 to 255, Signed characters have values from -128 to +127.

Type	Size (in Bytes)	Size (in Bits)	Range
$char\ or\ signed\ char$	1	8	$-128\ to\ 127$
$unsigned\ char$	1	8	$0\ to\ 255$

Floating Point Types

Floating point numbers are stored in 32 bits (on all 16bit and 32bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword $float$.

When the accuracy provided by a float number is not sufficient, the type $double$ can be used to define the number. This is also called long float. A $double$ data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. The double represents the same data type that float represents, but with greater precision. To extend the precision further, we may use long double which uses 80 bits.

Type	**Size (in Bytes)**	**Size (in Bits)**	**Range**
$float$	4	32	$3.4E-38$ to $3.4E+38$
$double$	8	64	$1.7E-308$ to $1.7E+308$
$long\ double$	10	80	$3.4E-4932$ to $1.1E+4932$

Note: *The size and range of Data Types vary from one machine to another. The above specified size and range of Data Types are on a **16-bit Machine**.*

Void Data Type

C also has a special data type called $void$, which indicates that any data type i.e., no data type does not describe the data items. The size of $void$ data type is 0 bytes.

Derived Data Types

These are also called secondary data types that include $arrays$, $structures$, $unions$ and $pointers$.

Type Declaration Instruction:

A variable can be used to store a value of any data type. This instruction is used to declare the type of variables being used in the program.
The syntax for declaring a variable is

$$\boldsymbol{data\_type\ variable1, variable2 \dots variablen;}$$

variable1, variable2,...variablen are the names of variables. Variables are separated by commas(,). Declaration statement must end with a semicolon(;).

E.g.: int count; int num,total;
char gender;
double ratio;
float rs;

Several subtle variations of the type declaration instruction are

1) While declaring the type of variable we can also initialize value
$int\ i = 10, j = 35;$

$float\ a = 1.5, b = 1.99 + 2.4 * 1.44;$

2) The order in which we define the variables is sometimes important sometimes not.
$int\ i = 10, j = 35;$ is same as $int\ j = 35, i = 10;$
However, $float\ a = 1.5, b = a + 3.1;$ is not same as $float\ b = a + 3.1, a = 1.5;$
(as we are trying to use **a** before defining it).

3) The following statements would work
$int\ a, b, c, d;$
$a = b = c = 10;$
& the following statements would not work
$int\ a = b = c = d = 10;$
as we are trying to use **b** (to assign to **a**) before defining it.

Declaring variable as constant

The value of certain variables might remain constant during the execution of a program which can be achieved with the qualifier **const** at the time of initialization.
E.g. $\boldsymbol{const\ int\ class\_size = 60};$
The keyword const tells the compiler that the value of the int variable class_size must not be modified by the program.

Declaring variable as volatile

Another qualifier **volatile** can be used to tell explicitly the compiler that a variable's value may be changed at any time by external sources (from outside the program).
E.g. $\boldsymbol{volatile\ int\ date};$
The value of date may be altered by some external factors. When we declare a variable as volatile, the compiler will examine the value of the variable each time whether any external alteration has changed the value and can be modified by its own program.

If the value must not be modified by the program while it may be altered by some other process, then we may declare the value as both const and volatile as

$$\boldsymbol{volatile\ const\ int\ location = 100};$$

Format Codes:

Format codes are coding characters used to represent the data types. They are needed must to print the output in a formatted manner. It is also called control string, or format specifiers or format strings.

Code	Description
%c	Single Character
%d	Decimal Integer
%i	Decimal, Hexadecimal or Octal Integer
%h	Short Integer

%e, %f, %g	Floating point value		
%ld	Long integer	%lu	Unsigned long int
%o	Octal Integer		
%s	String or sequence of Characters		
%u	Unsigned decimal integer		
%x	Unsigned Hexadecimal integer		

Basic Structure of C Program

The basic structure of the 'C' program consists of

```
Documentation Section
Link Section
Definition Section
Global Declaration Section
main () function section
{
        Declaration part;
        Executable part;
}
Sub program section
Function 1
Function 2
:
Function n
```

Documentation section

This section consists of a set of comment lines giving the name of the programmer, name of the program and other details, which the programmer would like to use later. It starts with '*' and ends with '*/'.

Link section

This section provides instructions to the compiler to link functions from system library. It is as "#include<stdio.h>".

Definition section

This section defines all the symbolic constants.

Global Declaration Section:

There are some variables that are used in more than one function, such variables are called global variables & are declared in the global declaration section.

main() function Section:

Every C program must have one 'main' function section. This section contains two parts i.e. declaration part and executable part. The declaration part declares all the variables used in the executable part. These two parts must appear between opening and closing braces {}.

Declaration & Execution Parts:

The program execution begins at opening brace and ends at the closing brace. All statements in the declaration and executable part ends with semicolon (;).

Sub Program functions:

The sub program functions contain all the user-defined functions that are called in the main function.

User defined functions are generally placed immediately after the main function.

The following are the **rules to write C Programs**:

1. All C statements must end with semicolon(;).
2. C is case-sensitive, i.e., upper and lower case characters are different. Generally, the statements are typed in lower case.
3. A C statement can be written in one line or it can split into multiple lines.
4. Every C program is a collection of one or more functions. There must be only function with the name as **main** as program execution starts and ends with main() function.
5. The function's body must be enclosed within braces. Braces must always match upon pairs i.e., every opening brace { must have a matching closing brace } to avoid confusions and eliminate errors.
6. To make programs easily understandable, comments can be added. Comments cannot be nested. For Example, /*Welcome to 'C', /*Programming*/*/ *(this cannot be used).* A comment can split into more than one line.

Execution of C Program in TurboC:

The following are the steps to be followed in writing and running a C program:

a) ***Creation of Source Program:***

Create a C program file in various C compilers that are available under MS-DOS, Turbo C Editor etc.

b) ***Compilation of the Program:***

Turbo C compiler is user friendly and provides integrated program development environment. Thus, selecting key combination can do compilation. That means press **Alt+F9** for compilation.

c) ***Program Execution:***

In Turbo C environment, the RUN option will do the compilation and execution of a program. Press **Ctrl+F9** for execution.

d) ***Result Display:***

In Turbo C environment, the output of the executed program can be viewed. Press **Alt+F5** to view the output.

Different types of files in TurboC

- In C language, every source file is saved with an extension of **“.c”**.
- The compiler automatically converts this source file into machine code at compile time and creates an executable file.
- The machine code is saved with an extension **“.obj”**.
- The executable file is saved with an extension of **“.exe”**.

Execution of C program in UNIX/LINUX systems

We use an editor in the UNIX system to create a C source program.

The following command is used to **compile** C program "helloworld.c"

- Compilation Command: **`gcc helloworld.c -o helloworld`**
  This command compiles the c program and generates an executable file "**helloworld**" for running the program. In the command if we don't specify "-o helloworld" it creates an executable file "**a.out**"

The following command is used to **execute** C program "helloworld.c"

- **./a.out** - to execute the program if executable file name is not specified.
- **./helloworld** - to execute the program if executable file name is specified.

I/O Functions

All most all the programming languages are designed to provide I/O activity. Input refers to the process through which data is supplied to the system. On the other hand, output refers to the provision which is made available by the computer to the user.

In C the input/output functions takes two forms they are: ***(1) Formatted (2) Unformatted***

Formatted I/O Functions

The Formatted input/output functions are standard functions capable to read and write all types of data values. They require conversion symbol to identify the data type. So they can be used for both reading and writing of all types. The formatted functions also return values after the execution. The formatted I/O functions supported by C are *printf()* and *scanf()*.

printf()

This is an output statement. To output data on to a screen, we use the standard output library function, represented by the word "printf" followed by the open and closing parentheses (). It is used to display the value of a variable or a message on the screen.

```
Syntax:
      printf("<message>");
      printf("<control string>", argument list separated with commas);
```

Example:

```
printf("This is C statement");
printf("The number is %d", a);
printf("The number %d is equal to %d", 10,10);
printf("The number %d is not equal to %d", x,y);
```

scanf()

This is an input statement. Data can be stored in the variables after accepting the values from the user through the keyword, by using a standard library function for input operation. This allows a program to get user input from the keyboard. This means that the program gets input values for variables from users.

```
Syntax:
scanf("<format code>",list of address of variables separated by commas);
```

Example:

```
scanf("%d", &a);
scanf("%d %c %f", &a, &b, &c);
```

1. To print a message "Hello World" on the screen

```
/*Program to print a message "Hello World" */
#include<stdio.h>
#include<conio.h>
main()
{
     clrscr();
     printf("Hello World");
}
Output:
Hello World
```

2. To Display Multiple Statements

```
/*Program to print Name and Address*/
#include<stdio.h>
#include<conio.h>
main()
{
   clrscr();
   printf("Name: Sachin Tendulkar");
   printf("\nQualification: Degree");
   printf("\nAddress: Mumbai")
   printf("\nWork: Cricket Player");
}
```

```
Output:
Name: Sachin Tendulkar
Qualification: Degree
Address: Mumbai
Work: Cricket Player
```

3. To Initialize int, char, float data types

```
/*Program to initialize int, char, float data types*/
#include<stdio.h>
#include<conio.h>
main()
{
   int n=78;
   float j=3.0;
   char x='y';
   clrscr();
   printf("Integer=%d\tFloat Value=%f\tCharacter=%c",n,j,x);
}
```

```
Output:
Integer=78    Float Value=3.0    Character=y
```

4. To accept the values of int, float, char data types and display them.

```
/*Program to accept values of int, char, float data types
Display them in the order of reading*/
#include<stdio.h>
#include<conio.h>
main()
{
   char x;
   int num;
   float j;
   clrscr();
   /*Accept the values for data types from user*/
   printf("Enter Character: ");
   scanf("%c",&x);
   printf("Enter Integer Value: ");
   scanf("%d",&num);
   printf("Enter Float Value: ");
   scanf("%f",&j);
   /*Display the accepted values*/
   printf("Integer=%d\tFloat Value=%f\tCharacter=%c",num,j,x);
}
```

```
Output:
Enter Character: a (Enter)
Enter Integer Value: 20 (Enter)
Enter Float Value: 100 (Enter)
Integer=20    Float Value=100.0    Character=a
```

Unformatted I/O Functions

A simple reading of data from keyboard and writing to I/O device, without any format is called unformatted I/O functions. The unformatted input/output functions only work with the character data type. They do not require conversion symbol for identification of data types. There is no need to convert the data. The unformatted functions also return values, and return value is always the same.

The unformatted I/O functions are classified into (1) Character I/O and (2) String I/O

Character I/O

These read and print single character.

(1) **getchar()**

It returns a single character accepted through the keyboard, after the confirmation key is pressed and can be assigned to the variable. The function does not require any parameter though a pair of empty parentheses must follow the word getchar.

```
Syntax: character_variable = getchar();
```

Example:

char c; c = getchar(); /*reads a single character*/

(2) **getch() & getche()**

These functions read any alphanumeric characters from the standard input device.

getch() is a standard library function used for single character input, but it does not wait for enter key and neither does it display the character on the screen.

```
Syntax: character_variable = getch();
```

getche() is a standard library functions used for single character input same as getch() function. It also does not wait for enter key but echoes on the screen. It can be used for getting the result directly on the screen.

```
Syntax: character_variable = getche();
```

Example:

char c; c = getche(); /*Displays the character entered on screen*/

c = getch(); /*does not display the character*/

(3) **putchar()**

It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character type variable. It must be expressed as a parameter to the function, enclosed in parentheses, followed by the word putchar.

Syntax: putchar(character_variable/expr);

Example:

char c; c = getchar(); putchar(c); /*prints the accepted character*/

(4) **putch()**

This function prints any alphanumeric character taken by the standard input device.

Syntax: putch(character_variable/expr);

Example: Program to accept characters and display them.

```
/*Program to accept characters and display*/
#include<stdio.h>
#include<conio.h>
main()
{
      char x,y,z;
      clrscr();
      printf("Enter 1st character: ");
      x = getchar();
      printf("Enter 2nd character: ");
      y = getche();
      printf("\nEnter 3rd character: ");
      z = getch();
      printf("\nFirst character is ");
      putchar(x);
      printf("\nSecond character is ");
      putch(y);
      printf("\nThird character is ");
      putchar(z);
}
```

Output:

```
Enter 1st character: k
Enter 2nd character: l
Enter 3rd character:
First character is k
Second character is l
Third character is ;
```

String I/O

In order to read and write string of characters the functions **gets()** and **puts()** are used. gets() function reads the string and puts() function takes the string as argument and writes on the screen.

WEEK – 3	
1	Operators in C, their Precedence and Associativity,
2	Arithmetic Expressions/Instructions
3	Type casting, Math.h functions
4	Sample C Programs

Operators in C

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations (calculations). C is extremely rich in built-in operators. Operators are used in program to manipulate data and variables. Some operators require two operands, while others act upon only one operand. C operators are classified as follows:

1) Arithmetic Operators
2) Relational Operators
3) Logical Operators
4) Assignment Operator
5) Increment & Decrement Operator
6) Conditional Operator
7) Bit wise Operators
8) Special Operators

Arithmetic Operators: C provides all basic arithmetic operators as listed below:

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Division (remainder after division)

Integer division truncates any fractional part. Modulo division produces the remainder of integer division.
E.g. sum = a+b; product = a*b; difference = a-b; quotient = a/b; remainder = a%b;
In the example, a & b are variables ad are known as operands.

Note: The modulo division operator (%) can't be used on float and double datatypes.

Relational Operators: Relational Operators are symbols that are used to test the relationship between 2 variables or between variable and a constant. We often compare two quantities & depending upon this relation take certain decisions.

Operator	**Meaning**
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

A simple relational expression contains only one relational operator and takes the following form:

$$ae1\ relational\ operator\ ae2$$

Here ae1 and ae2 are arithmetic expressions, which may be simple constants, variables or combination of them. The value of a relational expression is either *one* or *zero.* It is one if the specified relation is true and zero if the relation is false.
E.g. 10 < 20 is true, 20 < 10 is false.
These expressions are used in decision statements such as if and while to decide the course of action of a running program.

Logical Operators: C has the following three logical operators:

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

The logical operator && and || are used when we want to test more than one condition & make decisions.
E.g. a>b && x == 10

An expression of this kind will combine two or more relational expressions is termed as logical expression or compound relational expression. The logical expression given below is true if a>b is true and x == 10 is true. If either or both of them are false, the expression is false.

TRUTH TABLE			
op-1	**op-2**	**op-1 && op-2**	**op-1 || op-2**
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Logical NOT is used to reverse the truth value of its operand. (i.e. NOT F -> T)
An expression containing a logical operator is termed as a logical expression. A logical expression also yields a value of one or zero.

Assignment Operator: (=)
These are used to assign the result of an expression to a variable. C has a set of shorthand assignment operators of the form:

$$v\ op = exp;$$

where v is a variable, exp is an expression and op is a C arithmetic operator. The operator op= is known as shorthand assignment operator.
The above expression can be equivalent to $v = v\ op\ (exp)$;

E.g. x += y+1; -> x = x + (y+1);

Statement with simple assignment operator	**Statement with shorthand operator**
`a = a+1`	`a += 1`
`a = a-1`	`a -= 1`
`a = a*(n+1)`	`a *= (n+1)`
`a = a/(n+1)`	`a /= (n+1)`
`a = a%b`	`a %= b`

The use of shorthand assignment operators has 3 advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

Increment & Decrement Operators:
C has two very useful operators not generally found in other languages. These are increment and decrement operators: ++ and --.
The operator ++ adds 1 to the operand, while -- subtracts 1.

Pre/Post Increment/Decrement Operators
PRE means do the operation first followed by any assignment operation. POST means do the operation after any assignment operation.
++m; or m++; | --m; or m--;
++m; is equivalent to m=m+1; (or m+=1;) / --m; is equivalent to m=m-1; (or m-=1;)

While m++ and ++m mean the same when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.
E.g. (i) m = 5;
y = ++m; This statement results y and m = 6

Since the prefix operator first adds 1 to the operand and then the result is assigned to the variable on left.

(ii) m = 5;

y = m++; This statement results y = 5 and m = 6

Since the postfix operator first assigns the value to the variable on left then increments the operand.

We use increment and decrement statements in for and while loops extensively.

Conditional Operator:

A ternary operator pair "? :" is available in C to construct conditional expressions of the form:

$$exp1\ ?\ exp2 : exp3;$$

where exp1, exp2 and exp3 are expressions.

There operator ?: works as follows: exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

E.g. a=10;
b=15;
x=(a>b) ? a : b; In this, the x will be assigned with the value of b.

Bitwise Operators:

In C, operations on bits at individual levels can be carried out using Bitwise operators. These are used for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. These may not be applied to float or double.

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	One's complement

The **bitwise AND** does the logical AND of the bits in each position of a number in its binary form.

0 0 1 1 1 1 0 0 &
0 0 0 0 1 1 0 1
----
0 0 0 0 1 1 0 0 : Result

The **bitwise OR** does the logical OR of the bits in each position of a number in its binary form.

0 0 1 1 1 1 0 0 |
0 0 0 0 1 1 0 1
----
0 0 1 1 1 1 0 1 : Result

The **bitwise exclusive OR** performs a logical EX-OR function or in simple term adds the two bits discarding the carry. Thus result is zero only when we have 2 zeroes or 2 ones to perform on.

0 0 1 1 1 1 0 0 ^
0 0 0 0 1 1 0 1
----
0 0 1 1 0 0 0 1 : Result

The **one's complement (~)** or the bitwise complement gets us the complement of a given number. Thus we get the bits inverted, for every bit 1 the result is bit 0 and conversely for every bit 0 we have a bit 1.

Bit	**One's Complement**
0	1
1	0

~ 0 0 1 1 1 1 0 0 --> 1 1 0 0 0 0 1 1

Two **shift operators** shift the bits in an integer variable by a specified number of positions. The < < operator shifts bits to the left, and the > > operator shifts bits to the right.
The syntax for these binary operators is x << n and x >> n.

Each operator shifts the bits in x by n positions in the specified direction.

- For a **right shift**, zeros are placed in the n high-order bits of the variable;
- For a **left shift**, zeros are placed in the n low-order bits of the variable.

Here are a few examples:

Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).

Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).

Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).

Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

Special Operators:

C supports some operators of interest such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and ->).

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

E.g. value = (x=10,y=5,x+y);

First 10 is assigned to x then 5 is assigned to y & finally x + y i .e. which 15 is assigned to value .

Since comma operator has lowest precedence of all operators, the parentheses are necessary.

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

In while loops: while(c=getchar(), c!='10')

Exchanging values: t=x, x=y, y=t;

The **sizeof** is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

```
E.g.    m = sizeof(sum);
        n = sizeof(long int);
        k = sizeof(235L);
```

This operator is normally 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 execution of a program.

Precedence of Arithmetic operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High Priority	*, /, %
Low Priority	+, -

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered.

E.g. Consider a=9, b=12 and c=3

x = a - b/3+c*2-1;

```
        1> x = 9 - 12/3 + 3*2-1
        2> x = 9 - 4 + 3*2-1
        3> x = 9 - 4 + 6-1
        4> x = 5 + 6 - 1
        5> x = 11 - 1         6> x = 10
```

y = a - b/(3+c)*(2-1);

1> y = 9 - 12/6 * (2-1)

2> y = 9 - 12/6 * 1

3> y = 9 - 2 * 1

4> y = 9 - 2 5> y = 7

z = a - (b/(3+c)*2)-1;

1> z = 9 - (12/(3+3) * 2) - 1

2> z = 9 - (12/6 * 2) - 1

3> z = 9 - (2 * 2) - 1

4> z = 9 - 4 - 1

5> z = 5-1 6> z = 4

Operator Precedence and Associativity

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the highest level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as associativity property of an operator.

`a++ a--`	left to right
`!a ~a (`*`type`*`)a ++a --a`	right to left!
`a*b a/b a%b`	left to right
`a+b a-b`	left to right
`a>>b a<<b`	left to right
`a>b a>=b a<b a<=b`	left to right
`a==b a!=b`	left to right
`a&b`	left to right
`a^b`	left to right
`a|b`	left to right
`&&`	left to right
`||`	left to right
`a?b:c`	right to left
`=, +=, -=, *=, /=` `%=, <<=, >>=, &=` `|=, ^=`	right to left
`,`	left to right

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions.

Arithmetic Instruction

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names and constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, *, /, and %.

$$\boldsymbol{variable - name = expression\ [or]\ value};$$

E.g.
```
int a;
a=3200;
float kot, deta, alpha=9.2, beta=3.1256, gamma=100.0;
kot = 0.0056;
deta = alpha*beta/gamma + 3.2 * 2/5;
```

Here 2,5 and 3200 are integer constants and 3.2 and 0.0056 are real constants
kot, deta, alpha, beta and gamma are real variables.

The variables and constants together are called operands that are operated upon by the arithmetic operators and the result is assigned using the assignment operator, to the variable on the left-hand side.

C arithmetic instructions are of 3 types:

(1) ***Integer mode:*** This is an arithmetic instruction in which all operands are either integer variables or integer constants.

E.g.
```
int i, king, issac, noteit;
   i = i+1;
   king = issac*234 + noteit-7689;
```

(2) ***Real mode:*** This is an arithmetic instruction in which all operands are either real constants or real variables.

E.g.
```
float q, a, si, princ, anoy, roi;
   q = a + 23.123/4.5*0.344;
   si = princ*anoy*roi/100.0;
```

(3) ***Mixed mode:*** This is an arithmetic instruction in which some of the operands are integers and some of the operands are real.

E.g.
```
float si, princ, anoy, roi, avg;
   int a, b, c, num;
   q = a + 23.123/4.5*0.344;
   si = princ*anoy*roi/100.0;
   avg = (a+b+c+num)/4;
```

The execution of an arithmetic instruction: Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

Guidelines for Arithmetic Instructions

a) C allows only one variable on left hand side of =.
   i.e., x = k + l; is legal whereas k + l = x; is illegal.
b) An arithmetic instruction is often used for storing character constants in character variables.
   E.g.
   ```
   char a,b,d;
   a = 'F';
   b = 'G';
   d = '+';
   ```
   When we do this the ASCII values of the characters are stored in the variables. ASCII values are used to represent any character in memory.
c) Arithmetic operations can be performed on ints, floats and chars.
   ```
   char x,y;
   int z; x='a';
   y='b';
   z=x+y;
   ```
d) No operator is assumed to be present. It must be written explicitly.
   ```
   E.g.  a = c.d.b(xy)          usual arithmetic statement
         b = c*d*b*(x*y);       c statement
   ```
e) Unlike other high level languages, there is no operator for performing exponentiation operation.
   ```
   E.g.  a = 3**2;
         b = 3 ^ 2; statements are valid.
   ```

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

$$variable = expression;$$

Algebraic Expression	C expression
a x b - c	a * b - c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2 + 2x + 1$	3*x*x + 2*x + 1
$\frac{x}{y} + c$	x/y + c

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

```
E.g.    x = a*b-c;
        y = b/c*a;
        z = a-b/c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c and d must be defined before they are used in the expressions.

Rules for evaluation of expression

1. Parenthesized sub expression from left to right is evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub-expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
4. The associativity rule is applied when 2 or more operators of the same precedence level appear in a sub-expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parentheses assume highest priority.

Type Casting

Typecasting concept in C language is used to modify a variable from one date type to another data type. New data type should be mentioned before the variable name or value in brackets which to be typecast.

C type casting example program:

- In the below C program, 7/5 alone will produce integer value as 1.
- So, type cast is done before division to retain float value (1.4).

```
#include <stdio.h>
int main ()
{
    float x;
    x = (float) 7/5;
    printf("%f",x);
}
```

Output: 1.400000

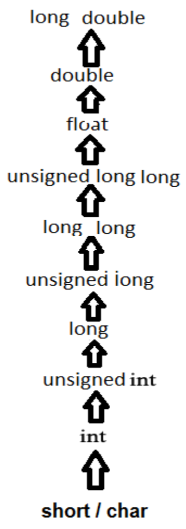
Note:

- It is best practice to convert lower data type to higher data type to avoid data loss.

- Data will be truncated when higher data type is converted to lower. For example, if float is converted to int, data which is present after decimal point will be lost.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values in a common type, C uses the rule that, in all expressions except assignments, any implicit type conversions made from a lower size type to a higher size type as shown below:



“math.h” functions

Mathematics is relatively straightforward library to use again. You must `#include<math.h>` and must remember to link in the math library at compilation:

```
   cc mathprog.c -o mathprog -lm
```

Math Functions

S.no	**Function**	**Description**	**Example**
1	**floor()**	This function returns the nearest integer which is less than or equal to the argument passed to this function.	floor of 5.100000 is 5.000000 floor of 5.900000 is 5.000000 floor of -5.400000 is -6.000000 floor of -6.900000 is -7.000000
2	**round()**	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from “.1 to .5”, it returns integer value less than the argument. If decimal value is from “.6 to .9”, it returns the integer value greater than the argument.	round of 5.400000 is 5.000000 round of 5.600000 is 6.000000
3	**ceil()**	This function returns nearest integer value which is greater than or equal to the argument passed to this function.	ceil of 5.400000 is 6.000000 ceil of 5.600000 is 6.000000 Ceil of 8.33=9

4	**sin()**	This function is used to calculate sine value.	The value of sin(0.314000) : 0.308866 The value of cos(0.314000) : 0.951106 The value of tan(0.314000) : 0.324744 The value of sinh(0.250000) : 0.252612 The value of cosh(0.250000) : 1.031413 The value of tanh(0.250000) : 0.244919 The value of log(6.250000) : 1.832582 The value of log10(6.250000) : 0.795880 The value of exp(6.250000) : 518.012817
5	**cos()**	This function is used to calculate cosine.	
6	**cosh()**	This function is used to calculate hyperbolic cosine.	
7	**exp()**	This function is used to calculate the exponential "e" to the xth power.	
8	**tan()**	This function is used to calculate tangent.	
9	**tanh()**	This function is used to calculate hyperbolic tangent.	
10	**sinh()**	This function is used to calculate hyperbolic sine.	
11	**log()**	This function is used to calculates natural logarithm.	
12	**log10()**	This function is used to calculates base 10 logarithm.	
13	**sqrt()**	This function is used to find square root of the argument passed to this function.	sqrt of 16 = 4.000000 sqrt of 2 = 1.414214
14	**pow()**	This is used to find the power of the given number.	2 power 4 = 16.000000 5 power 3 = 125.000000
15	**trunc()**	This function truncates the decimal value from floating point value and returns integer value.	truncated value of 16.99 = 16.000000 truncated value of 20.1 = 20.000000

Math Constants

The math.h library defines many (often neglected) constants. It is always advisable to use these definitions:

- HUGE - The maximum value of a single-precision floating-point number.
- M_E - The base of natural logarithms (e).
- M_LOG2E - The base-2 logarithm of e.
- M_LOG10E - The base-10 logarithm of e.
- M_LN2 - The natural logarithm of 2.
- M_LN10 - The natural logarithm of 10.
- M_PI - π.
- M_PI_2 - π/2.
- M_PI_4 - π/4.
- M_1_PI - 1/π.
- M_2_PI - 2/π.
- M_2_SQRTPI - $2/\sqrt{\pi}$.
- M_SQRT2 - The positive square root of 2.
- M_SQRT1_2 - The positive square root of 1/2.
- MAXFLOAT - The maximum value of a non-infinite single- precision floating point number.
- HUGE_VAL - positive infinity.

Sample C Programs

Input two numbers and compute all arithmetic operations

```
/*Program to accept 2 numbers and compute all arithmetic operations*/
#include<stdio.h>
#include<conio.h>
main()
{
    int num1,num2;
    clrscr();
    /*Accept two numbers from user*/
    printf("Enter first number: ");
    scanf("%d",&num1);
    printf("Enter second number: ");
    scanf("%d",&num2);
    /*Display values for arithmetic operators*/
    printf("Sum of 2 numbers: %d",num1+num2);
    printf("\nDifference of 2 numbers: %d", num1-num2);
    printf("\nProduct of 2 numbers: %d", num1*num2);
    printf("\nQuotient for %d/%d: %d", num1,num2,num1/num2);
    printf("\nRemainder for %d/%d: %d", num1,num2,num1%num2);
}
```

Output:

```
Enter first number: 52 (Enter)
Enter second number: 12 (Enter)
Sum of 2 numbers: 64
Difference of 2 numbers: 40
Product of 2 numbers: 624
Quotient for 52/12: 4
Remainder for 52/12: 4
```

Input radius, compute area, diameter, & circumference of the circle and display them.

```
/*Program to accept radius & calculate area, diameter and
circumference of circle*/
#include<stdio.h>
#include<conio.h>
main()
{
     /*Declare the variables*/
     int radius,diameter;
     float area,circumference;
     const float PI = 3.14; /*set variable PI to constant*/
     clrscr();
     /*Accept the value of radius*/
     printf("Enter circle radius: ");
```

```
      scanf("%d",&radius);
      /*Compute the area, diameter and circumference*/
      diameter = 2*radius;
      area = PI*radius*radius;
      circumference = 2*PI*radius;
      /*Display the results*/
      printf("Area of circle = %.2f",area);
      printf("\nDiameter of circle = %d",diameter);
      printf("\nCircumference of circle = %.2f",circumference);
      getch();
}
```

Output:

```
Enter circle radius: 5 (Enter)
Area of circle = 78.50
Diameter of circle = 10
Circumference of circle = 31.40
```

Swapping the values of two variables using third variable

```
/*Program to accept two numbers & swap the values*/
#include<stdio.h>
#include<conio.h>
main()
{
      /*Declare the variables*/
      int num1,num2,temp;
      clrscr();
      printf("Enter first number: ");
      scanf("%d",&num1);
      printf("Enter second number: ");
scanf("%d",&num2);
printf("Numbers before swapping: %d %d",num1,num2);
/*swapping the values of variables*/
temp = num1;
num1 = num2;
num2 = temp;
printf("Numbers after swapping: %d %d",num1,num2);
getch();
}
```

Output:

```
Enter first number: 6
Enter second number: 5
Numbers before swapping: 6 5
Numbers after swapping: 5 6
```

Program for swapping the values of two variables without using third variable

```
/*Program to accept two numbers & swap the values*/
#include<stdio.h>
#include<conio.h>
main()
{
      /*Declare the variables*/
      int num1,num2;
      clrscr();
      printf("Enter first number: ");
      scanf("%d",&num1);
      printf("Enter second number: ");
      scanf("%d",&num2);
      printf("Numbers before swapping: %d %d",num1,num2);
      /*swapping the values of variables*/
      num1 = num1 + num2;
      num2 = num1 – num2;
      num1 = num1 – num2;
      printf("Numbers after swapping: %d %d",num1,num2);
      getch();
}
```

Output:

```
Enter first number: 6
Enter second number: 5
Numbers before swapping: 6 5
Numbers after swapping: 5 6
```

Program to calculate total marks and percentage of a student for 5 subjects where marks of each subject should be greater than minimum pass marks (Ex: 35).

```
/*Program to accept marks and obtain total and percentage of marks*/
#include<stdio.h>
#include<conio.h>
main()
{
      int sub1,sub2,sub3,sub4,sub5,sum;
      long int studno;
      float total=500,percentage;
      clrscr();
      printf("Enter Student Number: ");
      scanf("%ld",&studno);
      printf("Enter SUBJECT1 marks: ");
      scanf("%d",&sub1);
      printf("Enter SUBJECT2 marks: ");
      scanf("%d",&sub2);
```

```
     printf("Enter SUBJECT3 marks: ");
     scanf("%d",&sub3);
     printf("Enter SUBJECT4 marks: ");
     scanf("%d",&sub4);
     printf("Enter SUBJECT5 marks: ");
     scanf("%d",&sub5);
     sum=sub1+sub2+sub3+sub4+sub5;
     percentage=(sum/total)*100;
     printf("============RESULT============\n");
     printf("STUDENT NUMBER: %ld",studno);
     printf("\nTOTAL MARKS OBTAINED FOR 500: %d",sum);
     printf("\nPERCENTAGE: %.2f",percentage);
     getch();
}
```

Output:

```
Enter Student Number: 1220610113
Enter SUBJECT1 marks: 90
Enter SUBJECT2 marks: 91
Enter SUBJECT3 marks: 95
Enter SUBJECT4 marks: 93
Enter SUBJECT5 marks: 89
============RESULT============
STUDENT NUMBER: 1220610113
TOTAL MARKS OBTAINED FOR 500: 458
PERCENTAGE: 91.60
```

To implement the concept of evaluating the expressions

```
/*Program to evaluate the expressions*/
#include<stdio.h>
main()
{
     int a=9,b=13,c=3;
     float x,y,z;
     x = a-b/3.0+c*2-1;
     y = a-(float)b/(3+c)*(2-1);
     z = a-((float)b/(3+c)*2)-1;
     printf("x = %f\t\ty = %f\t\tz = %f",x,y,z);
     getch();
}
```

Output:

```
x = 9.666667         y = 6.833333         z = 3.666667
```

WEEK – 4	
1	Control Statements (Conditional): If and its Variants
2	Switch (Break)
3	Sample C Programs

Control Statements (Conditional – Decision Making)

We have a number of situations where we may have to change the order of execution of statements based on certain conditions, or 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.

C language possesses such decision making capabilities and supports the following statements known as control or decision making statements.

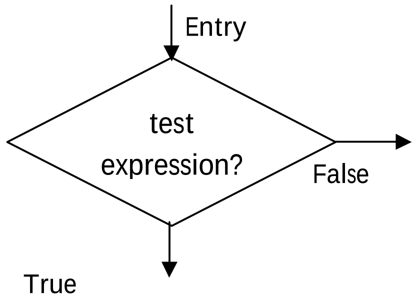
1. **if** statement
2. **switch** statement
3. **conditional operator** statement
4. **goto** statement

Decision making with 'if' statement

The **if** statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

$$if\,(test\ expression)$$

It allows the computer to evaluate the expression first and then depending on whether the value of expression (or condition) is true (1) or false (0), it transfers the control to a particular statement. This point of program has two paths to follow, one for the true condition and the other for the false condition.



Two-way Branching

Examples of decision making, using if statement are

1. if (bank balance is zero) borrow money
2. if(age is more than 60) person retires

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested.

1. Simple **if** statement
2. **if…else** statement
3. Nested **if…else** statement
4. **else if** ladder

Simple 'if' statement

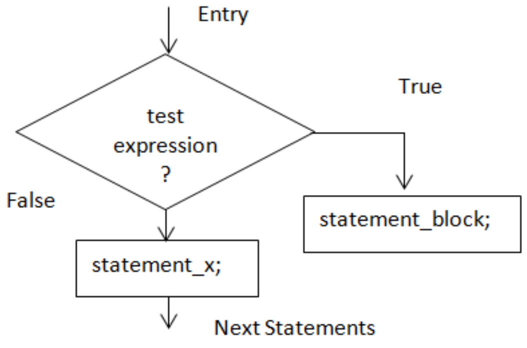
The general form of a 'simple if' statement is

Syntax:

```
if(test_expression)
{
     statement_block;
}
statement_x;
```

'statement_block' may be a single statement or a group of statements. If the test expression is true the 'statement_block' will be executed, otherwise the 'statement_block' will be skipped and the execution will jump to 'statement_x'.

Flowchart for Simple If



Example: To check whether student is passed or failed.

```
/*Program to check whether student is passed or failed*/
#include<stdio.h>
#include<conio.h>
main()
{
        int marks;
        clrscr();
        printf("Enter student marks: ");
        scanf("%d",&marks);
        if(marks>50)
          printf("Student Passed");
    if(marks<50)
          printf("Student Failed");
    getch();
}
```

Output:

```
(1) Enter student marks: 55
    Student Passed

(2) Enter student marks: 40
    Student Failed
```

If-Else Statement

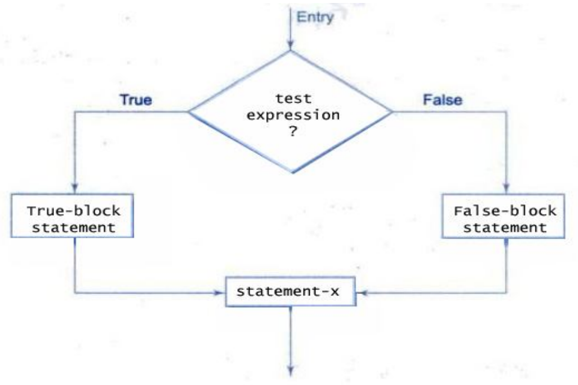
The if-else statement is an extension of the ‘simple if’ statement. The general form is

Syntax:

```
    if(test_expression)
    {
        true-block-statements;
    }
    else
    {
        false-block-statements;
    }
    statement_x;
```

If the test_expression is true, then the true-block-statement(s), immediately following the if statement are executed; otherwise the false-block-statement(s) are executed. In either case, either true-block-statements or false-block-statements will be executed, not both.

Flowchart for If Else



Example: Program to check whether given number is even or odd

```
/*Program to check whether given number is even or odd*/
#include<stdio.h>
#include<conio.h>
main()
{
        int num;
        clrscr();
        printf("Enter number: ");
        scanf("%d",&num);
        if(num%2 == 0)
          printf("%d is even number",num);
   else
          printf("%d is odd number",num);
   getch();
}
```

Output:

```
(1) Enter number: 53
   53 is odd number
(2) Enter student marks: 42
   42 is even number
```

Nested If… Else Statement

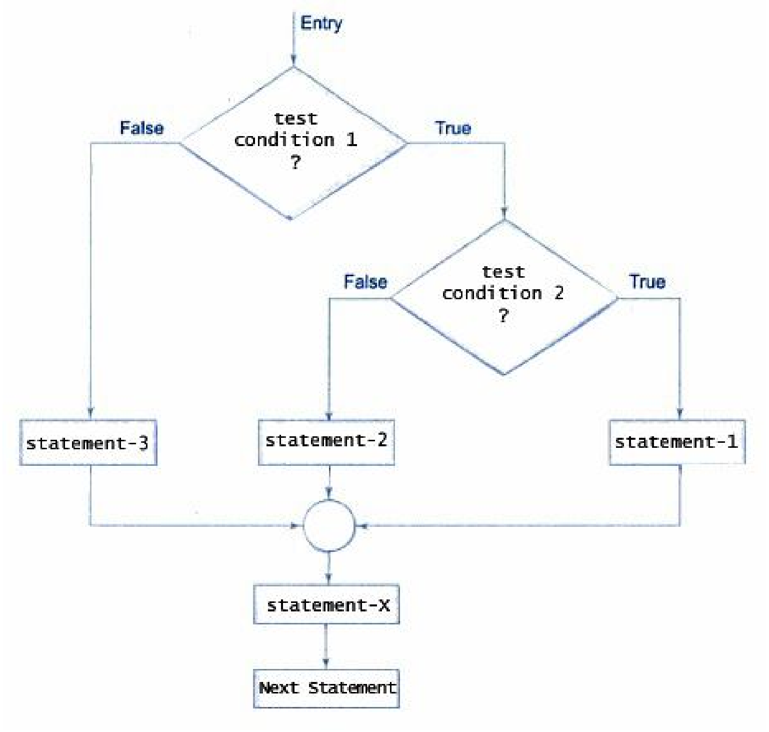
When a series of decisions are involved, we may have to use more than one if…else statement in nested form as follows:

Syntax:

```
    if(test_condition1)
    {
        if(test_condition2)
        {
            statement-1;
        }
        else
        {
            statement-2;
        }
    }
    else
    {
        statement-3;
    }
    statement-x;
```

If the test_condition1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the test_condition2 is true, the statement-1 will be executed otherwise statement-2 will be evaluated and then the control is transferred to the statement-x;

Flowchart for Nested If…Else



Example: Program to find the largest of three numbers

```
/*Program to find the largest of three numbers*/
#include<stdio.h>
#include<conio.h>
main()
{
        int a,b,c;
        clrscr();
        printf("Enter the three values: ");
        scanf("%d %d %d",&a,&b,&c);
        if(a>b && a>c)
           printf("%d is largest",a);
    else
           if(b>a && b>c)
              printf("%d is largest",b);
           else
              printf("%d is largest",c);
    getch();
}
```

Output:

```
Enter number: 5 6 7
7 is largest
```

Else If Ladder

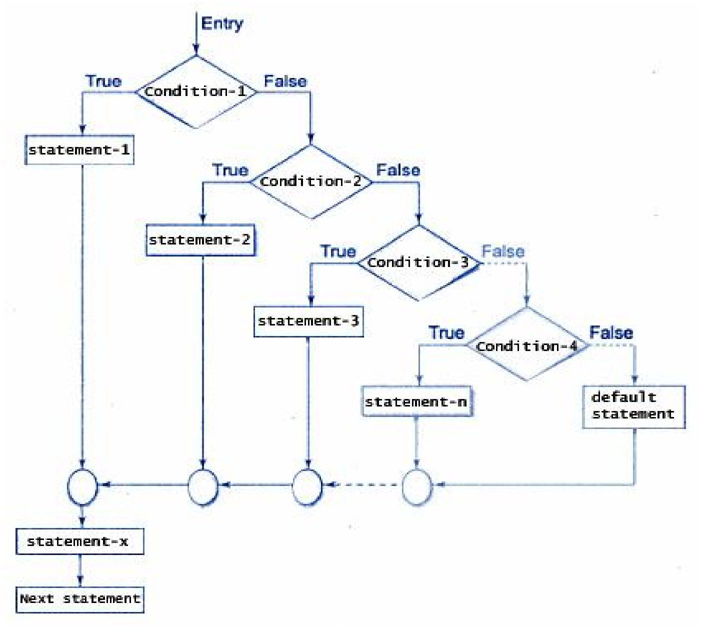
There is another way of putting if's together when multipath decisions are involved. A multipath decision is a chain of if's in which the statement associated with each else is an if. It takes the following general form:

Syntax:

```
    if(condition1)
          statement-1;
    else if(condition-2)
          statement-2;
    else if(condition-3)
          statement-3;
    ...
    ...
    ...
    else if(condition-n)
          statement-n;
    else
          default-statement;
    statement-x;
```

This construct is known as else if ladder. The conditions are evaluated from the top downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to statement-x. When all the n conditions become false, then the final else containing the default-statement will be executed.

The logic of execution for 'else if ladder statements' is shown in the flowchart below.



Example: Program to illustrate concept of else-if ladder to select color

```
/*Program to select color*/
#include<stdio.h>
#include<conio.h>
main()
{
        int n;
        clrscr();
        printf("Enter any number between 1 & 4 to select color: \n");
        scanf("%d",&n);
        if(n==1)
        {
                printf("You selected Red color");
        }
        else if(n==2)
    {
                printf("You selected Green color");
        }
    else if(n==3)
    {
                printf("You selected yellow color");
        }
        else if(n==4)
        {
                printf("You selected Blue color");
        }
        else
    {
                printf("No color selected");
        }
        getch();
}
```

Output:

```
(1)Enter any value between 1 & 4 to select color: 4
  You selected Blue Color
(2)Enter any value between 1 & 4 to select color: 1
  You selected Red Color
(3)Enter any value between 1 & 4 to select color: 5
  No color selected
```

SWITCH-CASE STATEMENT

When one of many alternatives is to be selected we can design a program using 'if' statement, to control the selection. However, the complexity of such programs in C number of alternatives increases. The program becomes difficult to read and follow.

The **switch** test or checks the values of given variable (or expression) against a list of case values and when a match is found a block of statements associated with that case is executed. The **switch** makes one selection when there are several choices to be made.

The general form of switch statement is

```
Syntax:
     switch(expression)
     {
          case value-1: block-1;
                        break;
          case value-2: block-2;
                        break;
          :
          :
          default: default-block;
                   break;
     }
     statement-x;
```

The expression is an integer expression or characters. Value-1, value-2, ... are constants or constant expressions and are known as case labels. Each of these values should be unique within a switch statement.

Block-1, block-2, ... are statements lists and may contain 0 or more statements. There is no need to put braces ({ }) around these blocks. Case labels end with a colon(:).

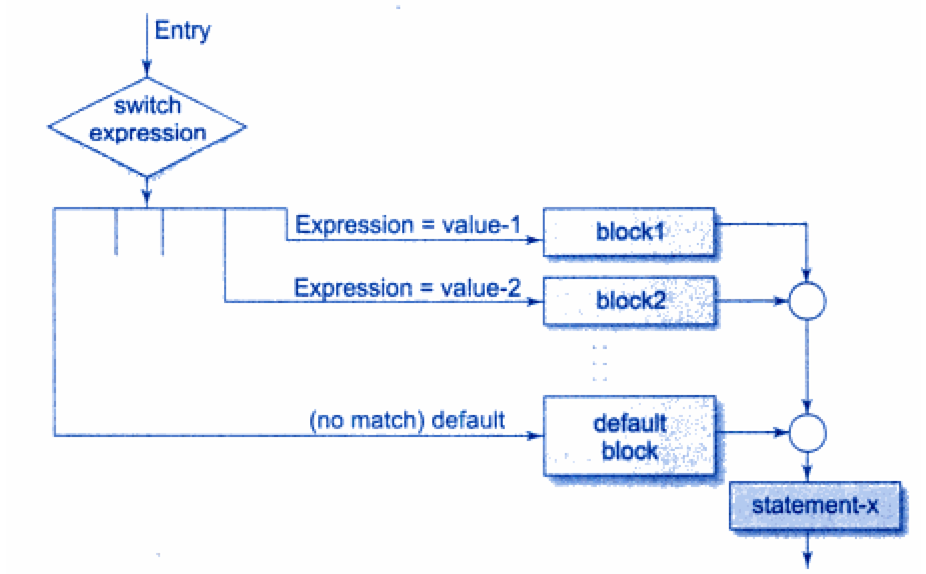
When a switch is executed the value of the expression is compared against the value (value-1, value-2, ...). If a case is found whose value of expression then block of statements that follows the case are executed.

The **break** statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch statement.

The **default** is an optional case, when present, it will be executed if the value of the expression does not match with any of the case values.

If not present, no action takes place and if all matches fail; the control goes to the statement-x.

The selection process of switch statement is illustrated in flow chart:



Example 1: Program to print words corresponding numbers below 9

```
/*Print words corresponding numbers below 9*/
#include<stdio.h>
#include<conio.h>
main()
{
      int n;
      clrscr();
      printf("Enter a number(0-9): ");
      scanf("%d",&n);
      switch(n)
      {
            case 0:    printf("Zero");
                       break;
            case 1:    printf("One");
                       break;
            case 2:    printf("Two");
                       break;
            case 3:    printf("Three");
                       break;
            case 4:    printf("Four");
                       break;
            case 5:    printf("Five");
                       break;
```

```
           case 6:    printf("Six");
                      break;
           case 7:    printf("Seven");
                      break;
           case 8:    printf("Eight");
                      break;
           case 9:    printf("Nine");
                      break;
           default:   printf("More than 9");
      }
      getch();
}
```

Output:

```
Enter a number (0-9): 3
Three

Enter a number (0-9): 10
More than 9
```

Algorithms for Conditional and Case Control

Example: Write an algorithm to find the largest of three numbers.

```
Step 1. Read the numbers a, b, c
Step 2. If a>b AND a>c then
Step 3. Print "a is the largest number"
Step 4. Else If b>a AND b>c then
Step 5. Print "b is the largest number"
Step 6. Else Print "c is the largest number"
Step 7. End of program.
```

Example: Write an algorithm to calculate pay salary with overtime.

[Salary depends on the pay rate and the number of hours worked per week. However, if you work more than 40 hours, you get paid time-and-a-half for all hours worked over 40.]

```
Step 1. Read hours and rate
Step 2. If hours ≤ 40 then
Step 3. Set salary as hours * rate
Step 4. Else
Step 5. Set salary as [40 * rate + (hours – 40) * rate * 1.5]
[salary = pay rate times 40 plus 1.5 times pay rate times (hours worked - 40)]
Step 6. Print salary
Step 7. End of program.
```

Pseudo code is an artificial and informal language that helps programmers develop algorithms. Pseudo code is a "text-based" detail (algorithmic) design tool. It is an algorithm written in English like language.

Pseudo-code Language Constructions:

Computation/Assignment

```
    set the value of "variable" to :"arithmetic expression" or
    "variable" equals "expression"
```

Input/Output

```
    get/read "variable", "variable", ...
    display "variable", "variable", ...
```

Conditional (dot notation used for numbering subordinate statements)

```
    6.   if  "condition"
             6.1   (subordinate) statement 1
             6.2    etc ...
    7.   else
             7.1   (subordinate) statement 2
             7.2   etc ...
```

Example : Write a pseudo-code to compute the final price of an item after figuring in sales tax.

```
Step 1. Begin
Step 2. get price_of_item
Step 3. get sales_tax_rate
Step 4. Set sales_tax to price_of_item times sales_tax_rate
Step 5. Set final_price to price_of_item plus sales_tax
Step 6. display final_price
Step 7. End.
```

Example : Write a pseudo code to check whether a student is passed or not.

```
Step 1. Begin
Step 2. Read marks
Step 3. If marks is greater than or equal to 60 then
Step 3.1. Print "Student Passed"
Step 4. Else
Step 4.1. Print "Student Failed"
Step 5. EndIf;
Step 6. End.
```

Algorithm to Find Volume of Sphere,Cone and Cylinder(using switch...case..default)

```
step 1: Start
step 2: Define pi<-3.14
step 3: Intilize 1<-1
step 4: check whether i=1 then go to step5 else go to step 24
step 5: Read x
step 6: check whether x=1,then go to step 7 else go to step 10
step 7: Read r
step 8: print 4/3*pi*r*r*r
step 9: break
step 10: check whether x=2,then go to step 11 else go to step 14
step 11: read r and h
step 12: print 1/3*pi*r*r*h
step 13: break
step 14: check whether x=3,then go to step 15 else go to step 18
step 15: read r and h
step 16: print 4/3*pi*r*r*r
step 17: break
step 18: check whether x=4,then go to step 19 else go to step 21
step 19: i=0
step 20: break
step 21: print invalid
step 22: break
step 23: go to step 4
step 24: Stop
```

Sample C Programs

1. To check whether number is +ve, -ve or zero

```
/*Program to check number is positive, negative or zero*/
#include<stdio.h>
#include<conio.h>
main()
{
int n;
clrscr();
printf("Enter a number: ");
     scanf("%d",&n);
     if(n>0)
          printf("Number is Positive");
     if(n<0)
```

```
          printf("Number is Negative");
     if(n==0)
          printf("Number is Zero");
}
```

Output:

```
Enter a number: -2
Number is Negative

Enter a number: 0
Number is Zero

Enter a number: 6
Number is Positive
```

2. To check two numbers are equal

```
/*Program to check whether the given numbers are equal*/
#include<stdio.h>
#include<conio.h>
main()
{
     int num1,num2;
     clrscr();
     printf("Enter 2 numbers: ");
     scanf("%d %d",&num1,&num2);
     if(num1==num2)
          printf("Both the numbers are equal");
     getch();
}
```

Output:

```
Enter 2 numbers: 6 6
Both the numbers are equal

Enter 2 numbers: 3 2
```

3. Check whether given character is vowel or consonant.

```
/*Program to check whether the given character is vowel or consonant */
#include<stdio.h>
#include<conio.h>
main()
{
     char x;
     clrscr();
     printf("Enter letter: ");
     x=getchar();
     if(x=='a'||x=='A'||x=='e'||x=='E'||x=='i'||x=='I'||x=='o'||x=='O'
          ||x=='u'||x=='U')
          printf("The character %c is a vowel",x);
     else
          printf("The character %c is a consonant",x);
     getch();
}
```

Output:

```
Enter letter: p
The character p is a consonant

Enter letter: a
The character a is a vowel
```

4. Program to calculate square of numbers whose least significant digit is 5.

```
/*Program to calculate square of numbers whose LSD is 5 */
#include<stdio.h>
#include<conio.h>
main()
{
     int s,d;
     clrscr();
     printf("Enter a Number: ");
     scanf("%d",&s);
     d=s%10;
     if(d==5)
     {
          s=s/10;
          printf("Square = %d %d",s*s++,d*d);
     }
     else
          printf("\nInvalid Number");
}
```

Output:

```
Enter a Number: 25
Square = 625

Enter a Number: 32
Invalid Number
```

5. To obtain the electric bill as per the charges below

No of Units Consumed	Rates (In Rs.)
500 and above	5.50
200-500	3.50
100-200	2.50
Less than 100	1.50

```
/*Program to obtain electric bill as per meter reading*/
#include<stdio.h>
```

```
#include<conio.h>
main()
{
     int initial,final,consumed;
     float total;
     clrscr();
     printf("Initial & Final Readings: ");
     scanf("%d %d",&initial,&final);
     consumed = final-initial;
     if(consumed>500)
          total=consumed*5.50;
else  if(consumed>=200 && consumed<=500)
          total=consumed*3.50;
     else if(consumed>=100 && consumed<=199)
          total=consumed*2.50;
     else if(consumed<100)
          total=consumed*1.50;
     printf("Total bill for %d units is %f",consumed,total);
     getch();
}
```

Output:

```
Initial & Final Readings: 1200 1500
Total bill for 300 units is 1050.000000

Initial & Final Readings: 800 900
Total bill for 100 units is 250.000000
```

6. To check whether letter is small, capital, digit or special symbol.

```
/*Program to check small,capital,digit or special*/
#include<stdio.h>
#include<conio.h>
main()
{
     char x;
     clrscr();
     printf("Enter character: ");
     scanf("%c",&x);
     if(x>='a' && x<='z')
          printf("Small letter");
     else if(x>='A' && x<='Z')
          printf("Capital letter");
     else if(x>='0' && x<='9')
          printf("Digit");
     else
          printf("Special Symbol");
     getch();
}
```

Output:

```
Enter character: *
Special Symbol

Enter character: T
Capital letter

Enter character: a
Small letter

Enter character: 6
Digit
```

7: Program to check whether a letter is vowel or consonant

```
/*Program to check whether given letter is vowel or consonant*/
#include<stdio.h>
#include<conio.h>
main()
{
     char ch;
     clrscr();
     printf("Enter Character: ");
     ch = getch();
     switch(ch)
     {
          case 'a':
          case 'A':
          case 'e':
          case 'E':
          case 'i':
          case 'I':
          case 'o':
          case 'O':
          case 'u':
          case 'U':  printf("Character %c is Vowel",ch);
                     break;
          default:   printf("Character %c is Consonant",ch);
     }
     getch();
}
```

Output:

```
Enter Character: a
Character a is Vowel
```

```
Enter Character: B
Character B is Consonant
```

8: Program to calculate Arithmetic Operations depending on operator.

```
/*Print words corresponding numbers below 9*/
#include<stdio.h>
#include<conio.h>
main()
{
      int a,b;
      char ch;
      clrscr();
      printf("Enter any arithmetic operator: ");
      scanf("%c",&ch);
      printf("Enter two numbers: ");
      scanf("%d %d",&a,&b);
      switch(ch)
      {
            case '+':  printf("\nThe sum is %d",a+b);
                       break;
            case '-':  printf("\nThe difference is %d",a-b);
                       break;
            case '*':  printf("\nThe product is %d",a*b);
                       break;
            case '/':  printf("\nThe quotient is %d",a/b);
                       break;
            case '%':  printf("\nThe remainder is %d",a%b);
                       break;
      default:   printf("Not Valid Operator");
      }
      getch();
}
```

Output:

```
Enter any arithmetic operator: *
Enter two numbers: 8 4
The product is 32
```

WEEK - 5	
1	Goto Statement, Control Statements (Looping): While,
2	Do..While, For Loop, Continue & Break (Unconditional)
3	Nested Loops,
4	Sample C Programs

GOTO Statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto might be desirable.

The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by colon. The label is placed immediately before the statement where the control is to be transferred.

The general forms of goto and label statements are shown below:

`goto label;`
`….`
`….`
`….`
`label;`
`statement;`
	`label;`
	`statement;`
	`….`
	`….`
	`….`
	`goto label;`

Forward Jump	**Backward Jump**

The label can be anywhere in the program either before or after the goto label; statement. During running of a program when a statement like **goto begin;** is met, the flow of control will jump to the statement immediately following the label **begin.** This happens unconditionally.

Example: To print Multiplication Table

```
/*Program to print Multiplication Table*/
#include<stdio.h>
#include<conio.h>
main()
{
     int a,i=1;
     clrscr();
     printf("Enter the value of a: ");
     scanf("%d",&a);
          printf("\nMultiplication Table for %d\n",a);
          printf("------------------------------\n");
     start:
          printf("%d x %d = %d\n",a,i,a*i);
          i=i+1;
          if(i<=10)
               goto start;
     getch();
}
```

Output:

```
Enter the value of a: 5

Multiplication Table for 5
------------------------------
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Advantage:

Using goto statement you can alter the normal sequence of the program execution so it gives the power to jump to any part of program.

Disadvantages:

It is always recommended not to use goto statement as this reduces the readability of the program. It becomes difficult to trace the control flow of a program, making the program logic complex to understand .Using goto statement is considered as poor programming practice. Any program written in C language can be written without the use of goto statement. So try to avoid goto statement as possible as you can.

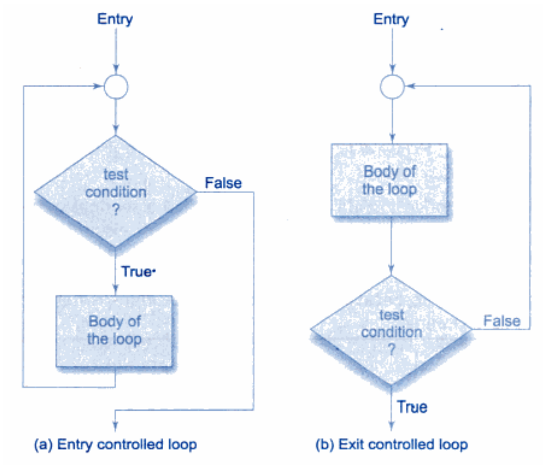
Looping

Loop is a mechanism through which you repeatedly execute a set of statements. In looping, a sequence of statements is executed until some condition for termination of the loop is satisfied.

A *program loop* therefore consists of two segments, one the body of the loop & other the control statement. The control statement tests certain conditions and then directs the repeated execution of statements contained in body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as **entry-controlled loop** or **exit-controlled loop**.

- In *entry-controlled loop*, the control conditions are tested before start of the execution. If conditions are not satisfied, then body of the loop will not be executed.
- In *exit-controlled loop*, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.



Loop Control Structures

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an infinite loop and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting & Initialization of a counter
2. Execution of the statements in the loop
3. Test for a specified condition for execution of the loop
4. Updating the counter

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides the following loop constructs:

(a) **while** statement
(b) **do..while** statement
(c) **for** statement

The WHILE Statement

- The simplest of all looping structures is the **while** (an entry-controlled loop) statement.
- The basic format of while statement is

```
Syntax:
     while(test_condition)
     {
          statement;
     }
```

test_condition	Is any valid C condition. Statement is repeatedly executed as long as condition is true. Once the condition is false, loop is terminated and control is transferred to the statement that is immediately after the loop.
statement	(Body of the loop) May be either a single statement or a compound statement.

- On exit, the program continues with the statement immediately after the body of the loop.

Example 1: To print the message 5 times

```
/*Print message 10 times*/
#include<stdio.h>
#include<conio.h>
main()
{
     int i=1;
     clrscr();
     while(i<=5)                    Output:
     {                              CPNM
          printf("\nCPNM");         CPNM
          i++;                      CPNM
}                                   CPNM
getch();                            CPNM
}
```

Example 2: To print 1 to 10 numbers.

```
/*Print 1 to 10 numbers*/          Output:
#include<stdio.h>                   1
#include<conio.h>                   2
main()                              3
{                                   4
     int i=1;                       5
     clrscr();                      6
     while(i<=10)                   7
     {                              8
```

```
          printf("\n%d",i);         9
          i++;                      10
 }
 getch();
 }
```

Example 3: To print 1 to 10 odd numbers.

```
 /*Print 1 to 10 numbers*/
 #include<stdio.h>
 #include<conio.h>
 main()
 {
      int i=1;
      clrscr();
      while(i<=10) {                Output:
          printf("\n%d",i);         1
          i+ = 2;                   3
 }                                  5
 getch();                           7
 }                                  9
```

Example 4: To print largest of the given numbers.

```
/*Program to display the largest of given numbers*/
#include<stdio.h>
main()
{
      int num;
      int largenum=0;
      clrscr();
      printf("Enter a number (0 to stop): ");
      scanf("%d",&num);
      while(num!=0)
      {
            if(num>largenum)
                  largenum = num;
            printf("Enter a number (0 to stop): ");
            scanf("%d",&num);
      }
      printf("Largest number is %d",largenum);
      getch();
}
```

```
Output:
Enter a number (0 to stop): 6
Enter a number (0 to stop): 9
Enter a number (0 to stop): 16
Enter a number (0 to stop): 98
Enter a number (0 to stop): 0
Largest number is 98
```

The DO…WHILE Statement

- When you need to execute statements at least for once irrespective of the result of the condition then you have to use do...while loop.
- Unlike while loop, in which condition is checked at the top of the loop; in do...while, condition is checked at the bottom.
- Do…while executes statements first and then checks condition. As the result statements are executed at least once as condition is not at all checked before the first iteration.
- The basic form of do…while is

```
Syntax:
   do
   {
       statements;
   }
   while(test_condition);
```

Braces { } are must for do…while. And we can have any number of statements between braces (body of the loop).

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

Example 1: To calculate sum of 10 natural numbers.

```
/*Calculate sum of 10 natural numbers*/
#include<stdio.h>
#include<conio.h>
main()
{
      int i=1,sum=0;
      clrscr();
      do
      {
            sum + = i;
            i++;
      }
      while(i<=10);
      printf("\nThe sum of 10 natural numbers is: %d",sum);
getch();
}
```

Output:
```
The sum of 10 natural numbers is: 55
```

Example 2: To print Multiplication table.
```
/*Program to print Multiplication Table*/
#include<stdio.h>
#define COLMAX 10
#define ROWMAX 10
main()
{
     int row,column,y;
     clrscr();
     row=1;
     printf("  MULTIPLICATION TABLE\n");
     printf("  -----------------------------------------\n");
     do
     {
          column=1;
          do
          {
               y=row*column;
               printf("%4d",y);
               column = column+1;
          }
          while(column<=COLMAX);
          printf("\n");
          row = row+1;
     }
     while(row<=ROWMAX);
     printf("  -----------------------------------------\n");
     getch();
}
```

Output:
```
  MULTIPLICATION TABLE
  -----------------------------------------
   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
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Example 3: To print Fibonacci Series.

```
/*Program to print Fibonacci Series 0,1,1,2,3,5,8,13,...*/
#include<stdio.h>
main()
{
	int n,n1=0,n2=1,n3,i=3;
	clrscr();
	printf("Enter number for series: ");
	scanf("%d",&n);
	printf("\nFibonacci Series: ");
	printf("\n%3d%3d",n1,n2);
	do
	{
		n3 = n1+n2;
		printf("%3d",n3);
		n1 = n2;
		n2 = n3;
		i++;
	} while(i<=n);
	getch();
}
```

Output:

```
Enter number for series: 10

Fibonacci Series:
  0  1  1  2  3  5  8 13 21 34
```

FOR Loop:

- This is another entry control loop.
- This integrates 3 basic ingredients of a loop (initialization, condition and incrementing).
- For loop is typically used to repeat statements for a fixed number of times.
- The basic form of for statement:

```
Syntax:
for(initialization;condition;updation)
{
	statement;
}
```

initialization	Executed only for once just before loop starts. Normally counter (variable used in loop) is initialized here.
condition	Is any valid C condition. As long as this is true statement is repeatedly executed.

updation	Executed after the statement is executed. Typically contains incrementing counter or decrementing counter as the case may be.
statement	(Body of the loop) This is repeatedly executed as long as condition is true. It may be a compound statement also.

Note: *All the above are optional. So any portion of the loop can be omitted. Though a portion is omitted, semicolon (;) after that must be given.*

The following is an example to display numbers from 1 to 10 using for loop.

```
      for(n=1;n<=10;n++)
            printf("%d\n",n);
```

This is in effect same as while loop used previously. But, as it combines initialization, condition and updation, it is more easier compared with while loop.

The following is another example for for loop where initialization and updation parts are omitted.

```
      /*this is to be terminated when 0 or negative number is given*/
      for(;n>0;)
            scanf("%d",&n);
```

Note: *In C language, for loop can be used in place of while loop and vice-versa. Both of them execute statements as long as the condition is true and terminate the loop once condition is false.*

The for statement

```
      for(; ;)
{
      statement;
      }
```

is an infinite loop. This can be terminated using a break statement or an exit() function.

Comma Operator

It is possible to have more than one expression in initialization and updation portions using (comma) operator. Comma operator is used to separate expressions.

The following is an example of for loop using comma operator:

```
      printf("\nn\tj");
for(n=0,j=10;n<j;n++,j--)
            printf("\n%d\t%d",n,j);
The output of the above example is
n       j
0       10
1       9
2       8
3       7
```

```
4       6
```

Example 1: To calculate factorial of any number

```
/*Program to calculate factorial of given number*/
#include<stdio.h>
main()
{
      int i,n;
      long int fact=1;
      clrscr();
      printf("Enter n value: ");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
            fact=fact*i;
      printf("Factorial of %d is %ld",n,fact);
      getch();
}
```

Output:

```
Enter n value: 8
Factorial of 8 is 40320
```

Example 2: Program to find number of even and odd numbers in the list

```
/*Program to find number of even and odd numbers in the list*/
#include<stdio.h>
#include<conio.h>
main()
{
      int i,n,num,ecount=0,ocount=0;
      clrscr();
      printf("Enter number of values: ");
      scanf("%d",&n);
      for(i=0;i<n;i++)
{
            printf("Enter a value: ");
            scanf("%d",&num);
            if(num%2==0) ecount++;
            else ocount++;
      }
      printf("Even count in the list of %d numbers is %d\n",n,ecount);
      printf("Odd count in the list of %d numbers is %d\n",n,ocount);
      getch();
}
```

Output:

```
Enter number of values: 10
Enter a value: 8
Enter a value: 6
Enter a value: 4
Enter a value: 12
Enter a value: 9
Enter a value: 2
Enter a value: 3
Enter a value: 1
Enter a value: 15
Enter a value: 98
Even count in the list of 10 numbers is 6
Odd count in the list of 10 numbers is 4
```

Example 3: Find out the sum of series $1^2 + 2^2 + ... + n^2$

```
/*Program to find the sum of series 12 + 22 + … + n2*/
#include<stdio.h>
#include<math.h>
main()
{
      int n, sum=0, i;
      clrscr();
      printf("Enter n value: ");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
      {
            sum = sum + pow(i,2);
      }
      for(i=1;i<=n;i++) {
            if(i!=n) printf("%d^2 + ",i);
            else printf("%d^2 = %d",i,sum);
      }
      getch();
}
```

Output:

```
Enter n value: 10
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2 = 385
```

Example 4: Find out the sum of series $1 + x^2/2! - x^3/3!+....$

```
/*Program to print sum of series 1 + x2/2! - x3/3!+....*/
#include<math.h>
main()
{
      int n,i,k,j,x,sign=1;
      int f;
      char ch;
```

```
    float sum=1;
    clrscr();
    printf("Enter number of terms: ");
    scanf("%d",&n);
    printf("Enter x value: ");
    scanf("%d",&x);
    k=1;
    for(i=1;i<n;i++)
    {
        j=k;
        f=1;
        while(j>0) {
            f = f*j;
            j--;
        }
        sum=sum+sign*((float)(pow(x,i))/f);
        k++;
        sign=sign*(-1);
    }
    printf("\n1");
    for(i=1;i<n;i++) {
        if(i%2==0) ch='-';
        else ch='+';
        printf(" %c %d^%d/%d!",ch,x,i,i);
    }
    printf(" = %f",sum);
    getch();
}
```

Output:

```
Enter number of terms: 5
Enter x value: 2

1 + 2^1/1! - 2^2/2! + 2^3/3! - 2^4/4! = 1.666667
```

break statement

This is used to terminate a loop. A loop can be terminated either when condition is false or when you execute break statement. When you have to terminate loop based on some other condition other than condition of the loop then you can use break statement.

Example 1: Program to implement break statement

```
/*Program to Implement break Statement*/
#include<stdio.h>
main()
{
    int i;
```

```
	clrscr();
	for(i=1; i<=10; i++)
	{
		printf("\n%d", i);
		if (i == 7)
			break;
	}
	getch();
}
```

Output:

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

Example 2: Program to display sum of 10 numbers or till 0 is given which comes first.

```
/*Program to display sum of given numbers*/
#include<stdio.h>
main()
{
	int sum=0,n,c;
	clrscr();
	for(c=1;c<=10;c++)
	{
		printf("Enter a number (0 to stop): ");
		scanf("%d",&n);
		if(n==0)
			break;
		sum += n;
	}
	printf("Sum = %d",sum);
	getch();
}
```

Output:

```
Enter a number (0 to stop): 5
Enter a number (0 to stop): 6
Enter a number (0 to stop): 4
Enter a number (0 to stop): 2
Enter a number (0 to stop): 8
Enter a number (0 to stop): 0
Sum = 25
```

continue statement

This is used to transfer control to the beginning of the loop from within the loop. It is used to skip the statements after continue statement and enter into next iteration of the loop.

Example: Program to Implement continue Statement

```
/*Program finds square of positive numbers only*/
#include <stdio.h>
main()
{
      int i, n, a, sq;
      clrscr();
      printf("\nHow many numbers you want to enter: ");
      scanf("%d", &n);
      for (i=1;i<=n; i++)
      {
            printf("\nEnter number: ");
            scanf("%d", &a);
            if(a<0)
                  continue;
            sq = a * a;
            printf("\nSquare = %d\n", sq);
      }
      getch();
}
```

Output:

```
How many numbers you want to enter: 3

Enter number: 2

Square = 4

Enter number: -1

Enter number: 6

Square = 36
```

Nested Loops

When a loop is placed inside another loop, it is called as nested loop. C allows nested loops. The inner loop or nested loop is executed for each repetition of the outer loop.

Example 1: Display numbers in the following format.

```
     1  2  3  4  5
     1  2  3  4  5
```

```
    1  2  3  4  5
    1  2  3  4  5
    1  2  3  4  5
#include<stdio.h>
main()
{
       int i,j;
       clrscr();
       for ( i = 1; i <= 5 ; i++)
       {
              printf("\n");
              for ( j = 1 ; j <= 5 ; j ++)
                     printf("%5d", j);
       }
       getch();
}
```

Example 2: Display numbers in the following format.

```
    1
    1  2
    1  2  3
    1  2  3  4
    1  2  3  4  5
#include<stdio.h>
main()
{
       int i,j;
       clrscr();
       for ( i = 1; i <= 5 ; i++)
       {
              for ( j = 1 ; j <= i ; j ++) {
                     printf("%5d", j);
              }
              printf("\n");
       }
       getch();
}
```

Example 3: Display numbers in the following format.

```
    1
    2  2
    3  3  3
    4  4  4  4
    5  5  5  5  5
#include<stdio.h>
main()
```

```
{
       int i,j;
       clrscr();
       for ( i = 1; i <= 5 ; i++)
       {
              for ( j = 1 ; j <= i ; j ++) {
                     printf("%5d", i);
              }
              printf("\n");
       }
       getch();
}
```

Example 4: Display numbers in the following format.

```
  1  1  1  1  1
  2  2  2  2
  3  3  3
  4  4
  5
```

```
#include<stdio.h>
main()
{
      int i,j;
      clrscr();
      for ( i = 1; i <= 5 ; i++)
      {
           for ( j = 5 ; j >= i ; j--) {
                printf("%5d", i);
           }
           printf("\n");
      }
      getch();
}
```

Example 5: Display numbers in the following format.

```
  5  4  3  2  1
  5  4  3  2
  5  4  3
  5  4
  5
```

```
#include<stdio.h>
main()
{
      int i,j;
      clrscr();
```

```
    for ( i = 1; i <= 5 ; i++)
    {
        for ( j = 5 ; j >= i ; j--) {
            printf("%5d", j);
        }
        printf("\n");
    }
    getch();
}
```

Example 6: Display numbers in the following format.

```
 5
 4  4
 3  3  3
 2  2  2  2
 1  1  1  1  1
```

```
#include<stdio.h>
main()
{
    int i,j;
    clrscr();
    for ( i = 5; i >= 1 ; i--)
    {
        for ( j = 5 ; j >= i ; j--) {
            printf("%5d", i);
        }
        printf("\n");
    }
    getch();
}
```

Example 7: Display numbers in the following format.

```
 *
 * *
 * * *
 * * * *
 * * * * *
```

```
#include<stdio.h>
main()
{
    int i,j,n;
    clrscr();
    n=5;
    for(i=0;i<n;i++)
    {
        for(j=0;j<=i;j++)
            printf(" *");
        printf("\n");
```

```
	}
	getch();
}
```

Example 8: Display Pascal's Triangle.

```
Enter the number of rows: 6
                 1
               1   1
             1   2   1
           1   3   3   1
         1   4   6   4   1
       1   5  10  10   5   1
```

```
#include<stdio.h>
main()
{
	int i,j,k,r,x;
	clrscr();
	printf("Enter the number of rows: ");
	scanf("%d",&r);
	i=1;
	for(k=0;k<r;k++)
	{
		for(j=30-3*k;j>0;j--)
			printf(" ");
		for(x=0;x<=k;x++)
		{
			if(x==0||k==0)
				i=i;
			else
				i=(i*(k-x+1)/x);
			printf("%6d",i);
		}
		printf("\n");
	}
	getch();
}
```

Algorithms for Looping Statements

Algorithm to find reverse of a given number using While Loop.

```
Step 1: Begin
Step 2: Display "Enter a number: "
Step 3: Read n
Step 4: Initialize rev to 0
Step 5: While "n!=0", do
Step 5.1: rem = n%10
Step 5.2: rev = (rev*10) + rem
Step 5.3: n = n/10
Step 6: EndWhile;
Step 7: Print "Reverse Number: ", rev
Step 8: End
```

Algorithm to print the Fibonacci Series using Do..While Loop.

```
Step 1: Begin
Step 2: Display "Enter a number: "
Step 3: Read n
Step 4: Initialize n1 to 0 and n2 to 1
Step 5: Print n1,n2
Step 6: Do
Step 6.1: n3 = n1+n2
Step 6.2: Print n3
Step 6.3: Set n2 to n1
Step 6.4: Set n3 to n2
Step 6.5: Increment i
Step 7: While "i<=n" goto Step 6
Step 8: EndDo;
Step 9: End
```

Algorithm to check whether given number is prime number or not.

```
Step 1: Begin
Step 2: Display "Enter a number: "
Step 3: Read n
Step 4: Initialize c to 0
Step 5: For i = 1 to n, do
        Step 5.1: If "n%i==0"
        Step 5.1.1: Increment c by 1
        Step 5.2: EndIf;
        Step 5.3: Increment i by 1
Step 6: EndFor;
Step 7: If "c<=2"
Step 7.1: Display n " is prime number"
Step 8: Else
Step 8.1: Display n " is not prime number"
Step 9: EndIf;
Step 10: End.
```

WEEK - 6	
1	Arrays
2	One Dimensional Array: Declaration and Initialization
3	Accessing Array Elements
4	Sample C Programs

What is an Array?

An Array is a collection of related data items which will share a common name. It is a **finite** collection of values of **similar** data type stored in adjacent memory locations that is accessed using one common name. Each value of the array is called as an element. Elements are accessed using index, which is a number representing the position of the element.

By finite we mean that, there are specific number of elements in an array and by similar we mean that, all elements in the array are of same type.

Declaring an array

An array is declared just like an ordinary variable but with the addition of number of elements that the array should contain.

A particular value is indicated by writing a number called 'index or subscript' number in brackets for after the array name.

Syntax:

```
type variablename[size];
```

The type specifies data type of element that will be contained in an array such as int, float, char, and double.

The size indicates the maximum number of elements that can be stored inside the array.

E.g. The following declaration declares n array of 20 elements where each element is of int type.

```
int marks[20];
```

Now, marks is the name of the array of 20 integers. Marks occupies 40 bytes, if int occupies 2 bytes (20*2).

Array can be of any variable type. The ability to use a single name to represent a collection to refer to an item by specifying the item number enables us to develop efficient programs.

TYPES:

Arrays in C Programming are of 3 types:

(1) One [or] Single Dimensional Array
(2) Two [or] Double Dimensional Array
(3) Multi Dimensional Array

ONE-DIMENSIONAL ARRAY

A list of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or one dimensional array.

For Example: If we want to represent a set of 5 numbers 35, 40, 20, 57, 19 by an array variable 'number' then we may declare the variable number as follows:

`int number[5];` and computer reverses 5 storage locations are shown below:

	number[0]
	number[1]
	number[2]
	number[3]
	number[4]

The values of an array element can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This could cause the array name to store the values as follows:

35	number[0]
40	number[1]
20	number[2]
57	number[3]
19	number[4]

The elements can be used in programs just like as any other 'C' variable.

It is not possible to either add new elements or delete existing elements after the array is created. In other words the size of the array is static and it cannot be changed at runtime.

How to access elements?

An array contains multiple elements. But the total array is accessed using one name. So to access array elements you have to use index or subscript and array name. The index is nothing but the number of the element. It identifies the position of the element in the array.
In C array index starts with 0 and ends at number of elements - 1.

E.g. marks[5] = 30; /*will store value 30 into 6th element*/

Subscripts: Subscripts of an array is an integer expression, integer constant or integer variables like 'i', 'n' etc. that refers to different elements in the array. Subscripts have a range, starting from '0' upto the "size of array-1".

Subscripts can also be reflected in for loops that print the array elements.

Consider a part of program of for loop

```
int number[8], n;
for(n=0;n<10;n++)
{
      printf("\nEnter Number: ");
      scanf("%d",&number[n]);
}
```

In the above program, 'n' is a subscript and n starts from 0 in the loop. The value at number[0] will be typed and stored in the first element of the array. This will continue until 'n' reaches the value of 10.

Initializing an array

It is possible to store values into an array during time of declaration.

    int a[5] = {10, 20, 30, 40, 50};

Values given in { } are placed in corresponding element of an array. If number of values given is more than the size, C displays an error "***Too many Initializers in Function...***". If number of values given is less than the size, then the number of elements then only the corresponding elements are filled.

In certain case, you can ignore the size of the array if you are initializing the array. C determines the size of the array from the number of values given in the initialization.

    int a[] = {20,30,40}; /*is an array of 3 elements*/

Sample C Programs

Example: Program to insert elements into an array and print the array elements.

```
#include<stdio.h>
#include<conio.h>
main()
{
     int a[5],i;
     clrscr();
     printf("Enter 5 elements into an array:\n");
     for(i=0;i<=4;i++) {
          scanf("%d",&a[i]);
}
printf("The 5 elements are:\n");
for(i=0;i<=4;i++)
     printf("%d\n",a[i]);
     getch();
}
```

Output:

```
Enter 5 elements into an array:
35
23
45
67
11
The 5 elements are:
35
23
45
67
11
```

Example: Take an array of 10 integers and accept values into it. Sort the array in descending order.

```
#include<stdio.h>
#include<conio.h>
main()
{
     int ar[10];
     int i, j, temp;
     clrscr();
     for (i = 0 ; i < 10 ; i++)
     {
          printf("Enter number for [%d] element: ",i);
          scanf("%d", &ar[i]);
     }
     /* sort array in descending order */
     for ( i = 0 ; i < 9 ; i++)
     {
     for ( j = i+1; j < 10 ; j ++)
     {
               if ( ar[i] > ar[j])
```

```
                {
                     /* interchange */
                     temp  =  ar[i];
                     ar[i] = ar[j];
                     ar[j] = temp;
                }
          } /* end of  j loop */
     } /* end of i loop */
     /* display sorted array */
     printf("\nSorted Numbers \n");
     for ( i = 0 ; i < 10 ; i++)
     {
          printf("%d\n", ar[i]);
     }
     getch();
}
```

Output:

```
Enter number for [0] element: 35
Enter number for [1] element: 67
Enter number for [2] element: 98
Enter number for [3] element: 12
Enter number for [4] element: 6
Enter number for [5] element: 58
Enter number for [6] element: 49
Enter number for [7] element: 23
Enter number for [8] element: 71
Enter number for [9] element: 85

Sorted Numbers
6
12
23
35
49
58
67
71
85
98
```

Example: Program to search an element in an array using Linear search (Sequential Search)

Linear search or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted.

```
#include<stdio.h>
int main(){
    int a[10],i,n,m,c=0;

    printf("Enter the size of an array: ");
    scanf("%d",&n);
    printf("Enter the elements of the array: ");
    for(i=0;i<=n-1;i++){
         scanf("%d",&a[i]);
    }
    printf("Enter the number to be search: ");
    scanf("%d",&m);
    for(i=0;i<=n-1;i++){
         if(a[i]==m){
             c=1;
             break;
         }
    }
    if(c==0)
         printf("The number is not in the list");
    else
         printf("The number is found");
    return 0;
}
```

Sample output:

```
Enter the size of an array: 5
Enter the elements of the array: 4 6 8 0 3
Enter the number to be search: 0
The number is found
```

Example: Program to search an element in an array using Binary search

The binary search begins by comparing the target value to value of the middle element of the sorted array.

- If the target value is equal to the middle element's value, the position is returned.
- If the target value is smaller, the search continues on the lower half of the array, or if the target value is larger, the search continues on the upper half of the array.
- This process continues until the element is found and its position is returned, or there are no more elements left to search for in the array and a "not found" indicator is returned.

```
#include<stdio.h>
int main(){
    int a[10],i,n,m,c=0,l,u,mid;
    printf("Enter the size of an array: ");
    scanf("%d",&n);
    printf("Enter the elements in ascending order: ");
    for(i=0;i<n;i++){
         scanf("%d",&a[i]);
    }
    printf("Enter the number to be search: ");
    scanf("%d",&m);
    l=0,u=n-1;
    while(l<=u){
         mid=(l+u)/2;
         if(m==a[mid]){
             c=1;
             break;
         }
         else if(m<a[mid]){
             u=mid-1;
         }
         else
             l=mid+1;
    }
    if(c==0)
         printf("The number is not found.");
    else
         printf("The number is found.");

    return 0;
}
```

Sample output:

```
Enter the size of an array: 5
Enter the elements in ascending order: 4 7 8 11 21
Enter the number to be search: 11
The number is found.
```

WEEK - 7	
1	Two Dimensional Array: Declaration and Initialization
2	Accessing Array Elements
3	Sample C Programs

TWO-DIMENSIONAL ARRAY

A list of items can be given one variable name using two subscripts and such a variable is called a double subscripted variable or two-dimensional array. So to access two-dimension array you have to have one row index and one column index.

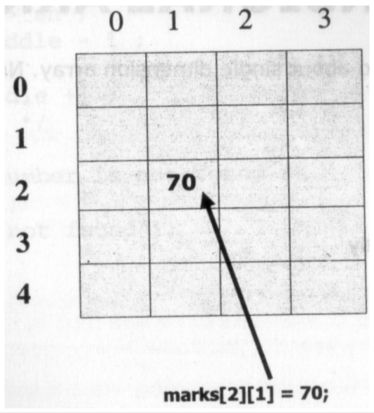
Declaration of Two-dimensional array

The general form of Two-dimensional arrays is

Syntax:

```
type array_name[row_size][column_size];
```

For example, to access 3rd row and 2nd column of marks array you would give:
marks[2][1] = 70;



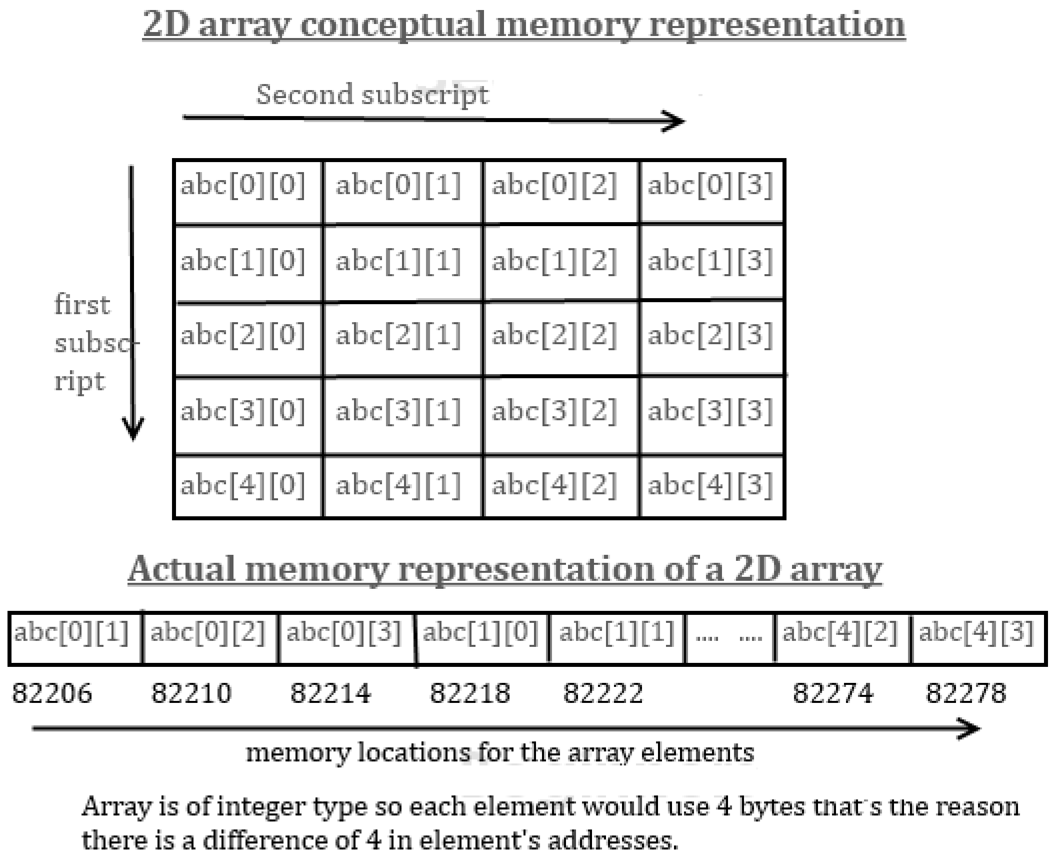
Initialization of 2D Array

There are many ways to initialize two Dimensional arrays –

```
int disp[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
OR
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Things which you must consider while initializing 2D array – You must remember that when we give values during one dimensional array declaration, we don’t need to mention dimension. But that's not the case with 2D array; you must specify the second dimension even if you are giving values during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = {1, 2, 3 ,4 }
/* Invalid because of the same reason mentioned above*/
int abc[2][] = {1, 2, 3 ,4 }
```


2D array conceptual memory representation
Second subscript
first subscript
abc[0][0] abc[0][1] abc[0][2] abc[0][3]
abc[1][0] abc[1][1] abc[1][2] abc[1][3]
abc[2][0] abc[2][1] abc[2][2] abc[2][3]
abc[3][0] abc[3][1] abc[3][2] abc[3][3]
abc[4][0] abc[4][1] abc[4][2] abc[4][3]
Actual memory representation of a 2D array
abc[0][1] abc[0][2] abc[0][3] abc[1][0] abc[1][1] .... abc[4][2] abc[4][3]
82206 82210 82214 82218 82222 82274 82278
memory locations for the array elements
Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

Sample C Programs

Example: Program to insert elements into array and display the array

```
#include<stdio.h>
#include<conio.h>
main()
{
      int a[2][2],i,j;
      clrscr();
      printf("Enter 4 elements into array: ");
      for(i=0;i<=1;i++)
      {
            for(j=0;j<=1;j++) {
                  scanf("%d",&a[i][j]);
            }
      }
      printf("The 4 elements are:\n\t");
      for(i=0;i<=1;i++)
      {
            for(j=0;j<=1;j++) {
                  printf("%d\t",a[i][j]);
            }
            printf("\n\t");
      }
      getch();
}
```

Output:

```
Enter 4 elements into array:
6
3
2
1
The 4 elements are:
        6       3
        2       1
```

Example: Program to print Multiplication Table

```
#include<stdio.h>
#include<conio.h>
#define ROWS 5
#define COLUMNS 5
main()
{
      int row,column,product[ROWS][COLUMNS];
      int i,j;
      clrscr();
      printf(" MULTIPLICATION TABLE \n\n");
      printf(" ");
      for(j=1;j<=COLUMNS;j++)
            printf("%4d",j);
      printf("\n");
      printf("-----------------------\n");
      for(i=0;i<ROWS;i++)
      {
            printf("   ");
            row=i+1;
            printf("%2d",row);
            for(j=1;j<COLUMNS;j++)
            {
                  column=j+1;
                  product[i][j] = row*column;
                  printf("%4d",product[i][j]);
            }
            printf("\n");
      }
      getch();
}
```

Output:

```
MULTIPLICATION TABLE

    1   2   3   4   5
-----------------------
    1   2   3   4   5
    2   4   6   8  10
    3   6   9  12  15
    4   8  12  16  20
    5  10  15  20  25
```

Example: Program to solve multiplication of 2 matrices

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int a[10][10],b[10][10],c[10][10],i,j,m,n,p,q,k;
      clrscr();
      printf("Enter order of matrix A for m & n\n");
      scanf("%d%d",&m,&n);
      for(i=0;i<m;i++)
      {
            for(j=0;j<n;j++)
            {
                  printf("Enter a[%d][%d] element: ",i,j);
                  scanf("%d",&a[i][j]);
            }
      }

      printf("Enter order of matrix B for p & q\n");
```

```
    scanf("%d%d",&p,&q);
    for(i=0;i<p;i++)
    {
          for(j=0;j<q;j++)
          {
                printf("Enter b[%d][%d] element: ",i,j);
                scanf("%d",&b[i][j]);
          }
    }

    if(n==p) {
          for(i=0;i<m;i++)
          {
                for(j=0;j<q;j++)
                {
                      c[i][j]=0;
                      for(k=0;k<m;k++)
                      c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
                }
          }

          printf("A MATRIX\n");
          for(i=0;i<m;i++)
          {
                for(j=0;j<n;j++)
                {
                      printf("%5d",a[i][j]);
                }
                printf("\n");
          }
          printf("B MATRIX\n");
          for(i=0;i<p;i++)
          {
                for(j=0;j<q;j++)
                {
                      printf("%5d",b[i][j]);
                }
                printf("\n");
          }
          printf("Elements after multiplication\n");
          for(i=0;i<m;i++)
          {
                for(j=0;j<q;j++)
                {
                      printf("%5d",c[i][j]);
                }
                printf("\n");
            }
        }
        else
            printf("Mutliplication Not Possible for given matrices!");
        getch();
}
```

Output:

```
Enter order of matrix A for m & n
2 2
Enter a[0][0] element: 1
Enter a[0][1] element: 2
Enter a[1][0] element: 3
Enter a[1][1] element: 4
Enter order of matrix B for p & q
2 2
Enter b[0][0] element: 2
Enter b[0][1] element: 3
Enter b[1][0] element: 4
Enter b[1][1] element: 5
A MATRIX
    1    2
    3    4
B MATRIX
    2    3
    4    5
Elements after multiplication
   10   13
   22   29
```

WEEK - 8	
1	Strings: Read & Write
2	"String.h" Predefined Functions
3	Sample C Programs

Strings Handling

A collection of characters is called as string. In the absence of string data type to store strings in C we have to simulate string using an array of characters. Any group of characters (except double quote sign) defined between double quotation marks is a constant string.

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings are:

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of a string

Declaring a String

An array of characters is also a collection of characters. So an array of characters is used to represent a string in C. For example, to store a name, which may be up to 50 characters, you would declare a string as follows:

```
/* name can contain up to 50 characters */
char name[50];
/* to store 10 names where each name can contain 20 characters */
char names[10][20];
```

How String is stored?

A string is stored as an array in the memory. However, when you declare a string of 20 characters, it doesn't always contain 20 characters. It may contain only 15 characters. So to identify the end of the actual characters in a string C is storing a null character (with ASCII code 0) at the end of the string.

Null character is written as '**\0**'. It is the character for ASCII code 0. As no other character contains ASCII code 0, when we encounter a null character in a string we can stop taking characters after that character.

name

J	O	H	N	S	O	N	P	\0	…

Note: Though you declare a string as of 50 characters, actually you can store only 49 characters. This is because one byte is used to store null character.

Initializing String Variables:

Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char text[9] = "WELCOME"; / char city[9] = "NEW YORK";
char text[9] = {'W', 'E', 'L', 'C', 'O', 'M', 'E'};
char text[9] = {'W', 'E', 'L', 'C', 'O', 'M', 'E', '\0'};
```

Input/output of Strings

You can read a string from keyboard. C has provided a conversion character **%s** for inputting and outputting string. You can read a string using **scanf()** & print using **printf()**.

The following code snippet is to read name of the user and display the same.

```
        char name[30];
        printf("What’s your name: ");
        scanf("%s",name);
        printf("Welcome %s",name);
```

There are few important points that you have to understand about reading strings using scanf()

- You must not precede string variable with & (ampersand). The reason will be evident to you once you understand pointers. Giving & before the string variable is a logical error. Compiler may not complain about it but program will not work.
- You can read the data from keyboard only up to first whitespace character (space or tab). That means if I want to enter my name, I can enter only Johson and not complete name “Johnson P D”. This is because of the fact that scanf() function assumes the end of the input for a field once it encounters a whitespace character.

Note: While using %s to read a string with scanf(), do not precede variable with & symbol.

Another important point about strings is, all standard functions such as scanf() and printf() take care of null character at the end of the string.

For instance, when you enter a string using scanf(), the function will automatically put a null at the end of the string. And in the same way while you are printing, printf() will take characters until null character is encountered.

String I/O

In order to read and write string of characters the functions gets() and puts() are used gets() function reads the string and puts() function takes the string as argument and writes on the screen.

(1) **gets()**

This function reads a sequence of characters entered through the keyword and is useful for interactive programming. Since a string has no predetermined length gets() needs a way to know to stop. It reads character until it reaches a new line character. To terminate input at the keyword, press the ‘Enter’ key.

```
Syntax: gets(variable);
```

(2) **puts()**

The function puts() writes a string argument onto the screen. The puts() can only output a string of characters. It takes less space and runs faster than printf().

```
Syntax: puts(variable);
```

Example: Program to print the accepted string.

```
/*Program to print accepted string*/
#include<stdio.h>
#include<conio.h>
main()
{
      char ch[30];
      clrscr();
      printf("Enter String: ");
      gets(ch);
      printf("\nEntered String: ");
      puts(ch);
}
```

Output:

```
Enter String: C is a good language.
Entered String: C is a good language.
```

The gets() and puts() functions offer simple alternatives with regard to the use of scanf() and printf() which are of reading and displaying strings respectively. These executes faster than printf() and scanf() due to the non-use of format specifies and occupies less space.

fflush(stdin):

The fflush(stdin) is a standard library function, which is used to clear the buffer, which it receives as a parameter.

```
Syntax: fflush(<std.buf.ref.>);
```

Program to read integer and character

```
/*Program to read integer and character*/
#include<stdio.h>
main()
```

```
{
     char ch;
     int a;
     clrscr();
     printf("Enter Integer value: ");
     scanf("%d",&a);
     fflush(stdin);
     printf("Enter character value: ");
     ch=getchar();
     printf("\nEntered integer = %d \tEntered character = %c",a,ch);
}
```

Output:

```
Enter Integer value: 22
Enter character value: k
Entered integer = 22    Entered character = k
```

String “string.h” Library

C provides a set of functions that performs operations on strings. These functions take a string and take actions such as reversing content of string, converting the string to upper case and return length of the string. The following are functions from string library. All functions are declared in **string.h** header file.

strlen() function

This function counts and returns the number of characters in a string. The length does not include a null character.

```
Syntax: len = strlen(string);
     where len is integer variable which receives the value of length
     of the string.
```

Example: Program to find length of the string using strlen() function.

```
#include<stdio.h>
#include<string.h>
main()
{
     char name[100];
     int length;
     printf("Enter a string: ");
     gets(name);
     length = strlen(name);
     printf("\nNumber of characters in the string is %d",length);
     getch();
}
```

Output:

```
Enter a string: peter
Number of characters in the string is 5
```

strlwr() function

This function converts all characters in a string from uppercase to lowercase.

```
Syntax: strlwr(string);
```

```
For Example: strlwr("EXFORSYS");
converts to exforsys.
```

strupr() function

This function converts all characters in a string from lowercase to uppercase.

```
Syntax: strupr(string);
```

```
For Example: strupr("exforsys");
converts to EXFORSYS.
```

Example: Program to convert lowercase to uppercase and uppercase to lowercase.

```
#include<stdio.h>
#include<string.h>
main()
{
      char name[100];
      clrscr();
      printf("Enter a string: ");
      gets(name);
      printf("\nUppercase String: ");
      puts(strupr(name));
      strlwr(name);
      printf("\nLowercase String: ");
      puts(name);
      getch();
}
Output:
Enter a string: peter
Uppercase String: PETER
Lowercase String: peter
```

strrev() function

This function reverses the characters in a string.

```
Syntax: strrev(string);
```

```
For Example: strrev("exforsys");
           reverses the characters to sysrofxe.
```

strcpy() function

C does not allow to assign the characters to a string directly as in the statement name="exforsys"; instead use the strcpy() function found in most compilers.

```
Syntax: strcpy(string1,string2);
For Example: strcpy(name,"exforsys");
          String exforsys is assigned to the string called name.
```

strcmp() function

In C we cannot directly compare the value of 2 strings in a condition like if(string1==string2)

Most libraries however contain the strcmp() function, which returns a zero if 2 strings are equal, or a non zero number (>0 or <0) if the strings are not same.

```
Syntax: strcmp(string1,string2);
```

```
string1 and string2 may be variables or constants. Some computers
return a negative value if string1 is alphabetically less than the
second and a positive if the string1 is greater than the second.
```

```
For Example:
strcmp("their","there");
     will return -9 which is the numeric difference between
     ASCII 'i' and ASCII 'r'.
strcmp("the","The");
     will return 32 which is the numeric difference between
     ASCII 't' and ASCII 'T'.
strcmp("hello","hello");
will return 0 as two strings are equal.
```

strcmpi() function

This function is same as strcmp() which compares 2 strings but not case sensitive.

```
Syntax: strcmpi(string1,string2);
For Example:
strcmpi("the","THE");
will return 0 as it ignores case of the string.
```

Example: Program to check whether given string is palindrome or not.

```
#include<stdio.h>
#include<string.h>
main()
{
     char name1[20],name2[20];
     clrscr();
     printf("Enter a string: ");
     gets(name1);
     strcpy(name2,name1);
```

```
      strrev(name2);
      printf("Reversed String is ",name2);
      if(strcmp(name1,name2) == 0)
            printf("String is Palindrome");
      else
            printf("String is Not Palindrome");
      getch();
}
Output:
Enter a string: peter
Reversed String: retep
String is not Palindrome
```

strcat() function

When you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The strcat() function joins 2 strings together. It takes the following form:

Syntax: `strcat(string1,string2);`

When the function strcat is executed string2 is appended to string1, the string at string 2 remains unchanged.

For Example:

```
      strcat(st1,"hello ");
      strcat(st2,"world");
      printf("%s",strcat(st1,st2));
From the above program segment of value of st1 becomes "hello world".
The string at st2 remains unchanged as "world".
```

Example: Program to sort an array of 10 strings

```
#include<stdio.h>
#include<string.h>
main()
{
      char names[10][20];
      int i,j;
      char temp[20];
      clrscr();
      printf("Enter array of strings:\n");
      for(i=0;i<10;i++)
            gets(names[i]);
      for(i=0;i<9;i++)
            for(j=i+1;j<10;j++)
            if(strcmp(names[i],names[j])>0)
            {
                strcpy(temp,names[i]);
                strcpy(names[i],names[j]);
                strcpy(names[j],temp);
          }
     printf("Sorted array of strings: \n");
     for(i=0;i<10;i++)
          puts(names[i]);
}
```

```
Output:
Enter array of strings:
santosh
anand
praneeth
manohar
naidu
hari
rajesh
aditya
julie
pavan
Sorted array of strings:
aditya
anand
hari
julie
manohar
naidu
pavan
praneeth
rajesh
santosh
```

strstr() function

This function returns the address (pointer) in string1 where string2 is starting in string1.

`Syntax: strstr(string1,string2);`

strchr() function

This function returns the address (pointer) of first occurrence of character **ch** in string.

`Syntax: strchr(string,ch);`

Sample C Programs for String Operations without using String Functions

Example: Program to read a line of text

```
#include<stdio.h>
main()
{
     char line[81],character;
     int c=0;
     clrscr();
     printf("Enter text. Press Enter at end\n");
     do {
          character = getchar();
          line[c] = character;
          c++;
     }
     while(character!='\n');
     c=c-1;
     line[c] = '\0';
     printf("\n%s\n",line);
     getch();
}
```

Output:

```
Enter text. Press Enter at end
This program reads a string and prints.

This program reads a string and prints.
```

Example: Program to Accept a string and display string in uppercase.

```
#include<stdio.h>
main()
{
       char st[20];
       int i;
       clrscr();
       /* accept a string */
       printf("Enter a string: ");
       gets(st);
       /* display it in upper case */
       for ( i = 0 ; st[i] != '\0';  i++)
              if ( st[i] >= 'a'  &&  st[i] <= 'z' )
                     putch( st[i] - 32);
              else
                     putch( st[i]);
       getch();
}
```

```
Output:
Enter a string: c programming
C PROGRAMMING
```

Example: Program to write string using %s format.

```
#include<stdio.h>
main()
{
      char country[15]="United Kingdom";
      clrscr();
      printf("\n\n");
      printf("------------------\n");
      printf("|%15s|\n",country);
      printf("|%5s|\n",country);
      printf("|%15.6s|\n",country);
      printf("|%.6s|\n",country);
      printf("|%15.0s|\n",country);
      printf("|%.3s|\n",country);
      printf("|%s|\n",country);
      printf("-----------------\n");
      getch();
}
```

```
Output:

------------------
| United Kingdom|
|United Kingdom|
|        United|
|United|
|               |
|Uni|
|United Kingdom|
------------------
```

Example: Program to find length of the string.

```
#include<stdio.h>
main()
{
      char str[20];
      int i = 0;
      clrscr();
      printf("\nEnter any string: ");
      gets(str);
      while (str[i] != '\0')
```

```
          i++;
     printf("\nLength of string: %d", i);
     getch();
}
```

Output:

```
Enter any string: ANIL NEERUKONDA
Length of string: 15
```

Example: Program to accept a string and display it in reverse.

```
#include<stdio.h>
main()
{
     char st[20];
     int i;
     clrscr();
     printf("Enter a string: "); /* accept a string */
     gets(st);
     /* get length of the string */
     for ( i = 0 ; st[i] != '\0';  i++);
     /* display it in reverse order */
     for ( i -- ; i >= 0 ; i --)
           putch(st[i]);
     getch();
}
```

Output:

```
Enter a string: HELLO WORLD
DLROW OLLEH
```

Example: Program to Concatenate of 2 Strings

```
#include<stdio.h>
#include<conio.h>
main()
{
     char string1[30], string2[20];
     int i, length=0, temp;
     printf("Enter the Value of String1: \n");
     gets(string1);
     printf("\nEnter the Value of String2: \n");
     gets(string2);
     for(i=0; string1[i]!='\0'; i++)
          length++;
     temp = length;
     for(i=0; string2[i]!='\0'; i++)
     {
          string1[temp] = string2[i];
          temp++;
     }
     string1[temp] = '\0';
     printf("\nThe concatenated string is:\n");
     puts(string1);
     getch();
}
```

Output:

```
Enter the Value of String1:
C Language

Enter the Value of String2:
is good

The concatenated string is:
C Language is good
```

Example: Program to Copy one string to another string.

```
#include <stdio.h>
#include <conio.h>
main()
{
     char string1[20], string2[20];
     int i;
     clrscr();
     printf("Enter the value of STRING1: \n");
     gets(string1);
     for(i=0; string1[i]!='\0'; i++)
          string2[i]=string1[i];
     string2[i]='\0';
     printf("\nThe value of STRING2 is:\n");
     puts(string2);
     getch();
}
```

Output:

```
Enter the value of STRING1:
c programs are cool

The value of STRING2 is:
c programs are cool
```

Example: Program to Compare two given strings.

```
#include<stdio.h>
main()
{
	char string1[15],string2[15];
	int i,temp = 0;
	clrscr();
	printf("Enter the string1 value:\n");
	gets(string1);
	printf("\nEnter the String2 value:\n");
	gets(string2);
	for(i=0;string1[i] != ‘\0’;i++) {
		if(s1[i] == s2[i])
			temp = 1;
		else
			temp = 0;
	}
	if(temp==1)
		printf("Both strings are same.");
	else
		printf("Both strings not same.");
	getch();
}
```

Output1:

```
Enter the string1 value:
c programs

Enter the String2 value:
cprograms

Both strings not same.
```

Output2:

```
Enter the string1 value:
c programs

Enter the String2 value:
c programs

Both strings are same.
```

Example: Program to print alphabet set in decimal and character form.

```
#include<stdio.h>
main()
{
	char c;
	clrscr();
	printf("\n\n");
```

```
     for(c=65;c<=122;c=c+1)
     {
          if(c>90 && c<97)
               continue;
          printf("%5d - %c",c,c);
     }
     printf("\n");
     getch();
}
Output:
65 - A   66 - B   67 - C   68 - D   69 - E   70 - F   71 - G   72 - H
73 -I   74 - J   75 - K   76 - L   77 - M   78 - N   79 - O   80 - P
81 - Q   82 - R   83 - S   84 - T   85 - U   86 - V   87 - W   88 - X
89 - Y   90 - Z   97- a   98 - b   99 - c  100 - d  101 - e  102 - f
103 - g  104 - h  105 - i  106 - j  107 - k  108 - l  109 - m  110 - n
111 - o  112 - p  113 - q  114 - r  115 - s  116 - t  117 - u  118 - v
119 - w  120 - x  121 - y  122 - z
```

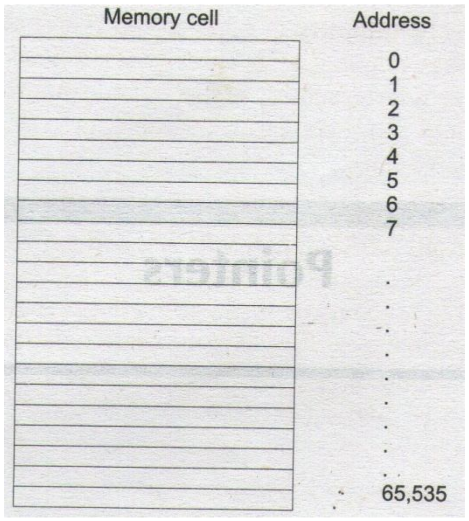
UNIT - 4

WEEK - 9	
1	Pointers: Declarations, Types, Pointers to Arrays
2	Pointers to Character Strings
3	Pointers to Pointers, Array of Pointers

Pointers

The most intimidating topic in C language is ***pointers***. To understand the concept of pointers we require complete disciplined approach and a lot of concentration and a few pictures of memory.

Computer's use their memory for storing the instructions of a program as well as the values of variables that are associated with it. The computer's memory is a sequential collection of storage cells. Each cell commonly known as a byte, has a number called address associated with it. The addresses are numbered starting from zero. The last address depends on memory size. Whenever we declare a variable, the system allocates somewhere in a memory an appropriate location to hold the value of the variable. Since every byte has a unique address number, this location will have its known address number.



Memory Organization

Consider the following statement, $int\ a\ =\ 179$; this statement instructs the system to find a location for the integer variable 'a' and puts the value 179 in that location.

$$\begin{tikzcd}
\text{Variable} \arrow[r] & a \\
\text{Value} \arrow[r] & \boxed{179} \\
\text{Address} \arrow[r] & 5000
\end{tikzcd}$$

We may have to access to the value 179 by using either the name ‘a’ or address ‘5000’.

Since, memory addresses are simple numbers; they can be assigned to some variables which can be stored in memory like any other variable. Such variables that hold memory addresses are called **pointers**.

What is Pointer?

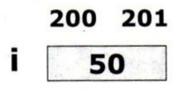
A pointer is a variable that contains an address which is a location of another variable in memory. A pointer enables us to access a variable that is defined outside the function. Pointers reduce the length and complexity of a program. They increase the execution speed.

Understanding a variable

A variable contains a name, value and address of memory location where it stores its value.

Consider the following variable declaration and memory picture. $int\ i$ = 50;

The following would be the memory picture after the memory is allocated to variable ‘i’.



Note: All addresses used in this chapter are imaginary. They are used only to let you understand the concept better. Picture doesn’t show how exactly the value is stored in memory and that is not important in this case.

So looking at the figure, you would understand that integer variable ‘i’ is allocated two bytes in memory. First byte is at address 200 and second byte at 201. The value stored in the variable is 50.

Note: Every memory location contains an address. The address of the variable is always the address of the first byte of the space allocated to a variable.

Pointer Variable:

A pointer variable or a pointer is similar to a variable but where a variable stores data; a pointer variable stores address of a memory location. So a pointer does contain all attributes of a variable but what is stored in pointer is interpreted as an address of a memory location and not as data.

Declaring a pointer

A pointer variable (or pointer) is declared same as a variable but contains a * before the name.

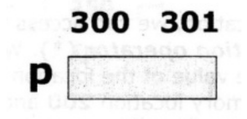
Syntax: $\boldsymbol{datatype * pointer\_name}$;

This tells the compiler three things about the variable ptr_name:

- ❑ The $*$ tells that the variable ptr_name is a pointer variable.
- ❑ The ptr_name needs a memory location.

- ptr_name points to a variable of a data type.

The following is an example of a pointer variable. **int `*p;`** declares the variable p as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer. Just like a normal variable even a pointer variable contains memory location.



Similarly the statement, **float `*x;`** declares x as a pointer to a floating point variable.

Note: A pointer always occupies 2 bytes in memory. This is because to store an address we need only 2 bytes.

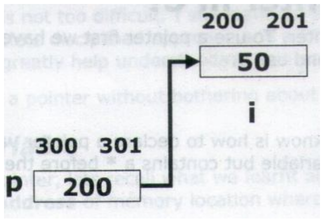
Storing address into pointer

A pointer is used to contain an address, to store the address of variable into pointer using address operator (&). The operator & immediately preceding a variable returns the address of the variable associated with it.

When a pointer contains the address of a variable, then the pointer is said to be pointing to that variable. For example, in the following code, pointer **p** is pointing to variable **i.**

```
int i=50,*p;
        p = &i;
```

The memory image for the above assignment is shown in the figure below:



When p contains the address of **i** then it is read as ***p is pointing to i.***

Accessing a Variable through its pointer

To access the value of the variable using pointer (that contains address of a variable), this is done by using another unary operator * (asterisk), usually known as the indirection operator. When a pointer points to a memory location we can access the value of that memory location using a special operator called **indirection operator (*)**.

When a pointer variable is used with indirection operator then you get the value of the location to which pointer points. So *p will return **50** because p is pointing to memory location 200 and that is the location of variable i, where value 50 is stored.

For Example,

```
int quantity,*p,n;
quantity = 150;
```

```
p = &quantity;
n = *p;
```

- The first line declares quantity and n as integer variables and p as a pointer variable of an integer type.
- The second line assigns the value 150 to quantity.
- The third line assigns the address of quantity to pointer variable p.
- The fourth line contains the indirection operator. When the operator ***** is placed before a pointer variable in an expression (on the right hand side of equal sign), the pointer returns value of the variable of which the pointer value is the address. So, ***p** returns the value of the variable quantity, as p is the address of quantity. Thus the value at address p will be assigned to n, where n value would be 150.

Example: Program to illustrate the concept of pointers.

```
/*Program to Illustrate the Concept of Pointers*/
#include <stdio.h>
main()
{
     int a = 10;
     int *p;
     p = &a;
     clrscr();
     printf("\nAddress of a: %u", &a);
     printf("\nAddress of a: %u", p);
     printf("\nAddress of p: %u", &p);
     printf("\nValue of p: %d", p);
     printf("\nValue of a: %d", a);
     printf("\nValue of a: %d", *(&a));
     printf("\nValue of a: %d", *p);
     getch();
}
```

Output:

```
Address of a: 65494
Address of a: 65494
Address of p: 65496
Value of p: 65494
Value of a: 10
Value of a: 10
Value of a: 10
```

Example: Program to illustrate the use of indirection operator.

```
/*Program to Illustrate the use of Indirection operator*/
#include <stdio.h>
main()
{
     int x,y;
     int *p;
```

```
      clrscr();
      x = 10;
      p = &x;
      y = *p;
      printf("\nValue of x: %d", x);
      printf("\n%d is stored at address %u", x,&x);
      printf("\n%d is stored at address %u", *p,p);
      printf("\n%d is stored at address %u", y,&*p);
      printf("\n%u is stored at address %u", p,&p);
      printf("\n%d is stored at address %u", y,&y);
      *p = 25;
      printf("\nValue of x: %d", x);
      getch();
}
```

Output:

```
Value of x: 10
10 is stored at address 65492
10 is stored at address 65492
10 is stored at address 65492
65492 is stored at address 65496
10 is stored at address 65494
Value of x: 25
```

Significance of data type

When you declare an integer variable using int i; then i occupies 2 bytes, and float f; would occupy 4 bytes. So the data type determines how many bytes are given to variable.

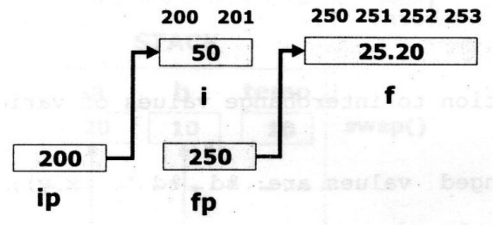
We must ensure that the ***pointer variables always point to the corresponding type of data***. For example,

```
float a,b;
int x,*p;
p = &a;
b = *p;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

To understand the significance of data type in pointer declaration, you need to understand what happens when we use * operator. The following example will explain the process involved in using a pointer.

```
int i=50;
int *ip;
float f=25.20;
float *fp;
ip=&i;
fp=&f;
printf("%d %f",*ip,*fp);
```



In the above example, **ip** contains the starting address of **i**, which is 200. But when we use ***ip** to get the value of location pointed by **ip** we get the value stored in locations 200 and 201.

Here are the steps that are taken when a pointer is used with indirection operator (*).

- First, the value of pointer variable is taken {***ip = *(200)**}
- Then it goes to location with that address.
- From that address it takes value from n bytes, where n depends on the data type of the pointer. In case of **ip** it is int, so it will be two bytes starting from the given address. {***ip = *(200) = 50**}

The same will be the process for ***fp**, but it will take value from 4 bytes (250 to 253)

Pointer Expressions:

Like other variables, pointer variables can be used in expressions. For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid:

```
y = *p1 * *p2; same as (*p1)*(*p2)
sum = sum + *p1;
z = 5 * - *p2/ *p1; same as (5 * (-(*p2)))/(*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and *. The following is wrong:

```
z = 5 * - *p2/*p1;
```

As the symbol /* is considered as beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another: p1 + 4, p2 – 2 and p1 – p2 are all allowed. If p1 and p2 are both pointers to the same array then p2 – p1 gives the number of elements between p1 and p2.

We may also use short-hand operators with the pointers.

```
p1++;
--p2;
sum += *p2;
```

In addition to arithmetic operators, pointers can also be compared using the relational operators. The expressions such as p1>p2, p1 = -p2 and p1 != p1 are allowed. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as p1/p2 or p1*p2 or p1/3 are not allowed. Similarly, two pointers cannot be added. That is, p1+p2 is illegal.

Example: Program to illustrate the use of pointers in arithmetic operators.

```
/*Program to Illustrate the use of pointers in arithmetic operators*/
#include <stdio.h>
main()
{
      int a,b,*p1,*p2,x,y,z;
      clrscr();
      a=12;
      b=4;
      p1=&a;
      p2=&b;
      x=*p1 * *p2-6;
      y=4 * -*p2/ *p1 + 10;
      printf("Address of a=%u\n",p1);
      printf("Address of b=%u\n",p2);
      printf("\n");
      printf("a=%d b=%d\n",a,b);
      printf("x=%d y=%d\n",x,y);
      *p2 = *p2+3;
      *p1 = *p2-5;
      z = *p1 * *p2-6;
      printf("\na=%d b=%d z=%d",a,b,z);
      getch();
}
```

Output:

```
Address of a=65492
Address of b=65494

a=12 b=4
x=42 y=9

a=2 b=7 z=8
```

Pointer Increment and Scale Factor – Address Arithmetic

We have seen that the pointers can be incremented like p1 = p2 + 2; p1 = p1 + 1; and so on. However, an expression like **p1++;** will cause the pointer p1 to point to the next value of its type.

For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1=p1+1**, the value of p1 will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the scale factor.

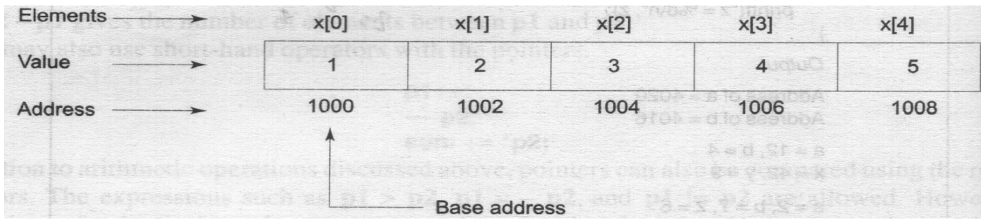
The number of bytes used to store various data types depends on the system and can be found by making use of the sizeof operator.

POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array x as follows:

int x[5] = {1,2,3,4,5};

Suppose the base address of x is 1000 and assuming that each integer requires 2 bytes, the five elements will be stored as follows:



The name x is defined as a constant pointer pointing to the first element x[0] and therefore the value of x is 1000 i.e., x = &x[0] = 1000.

Similarities between pointer and array

A pointer and an array are similar in some respects. Array name x contains address of the first element of array. **&x[0] = = x**

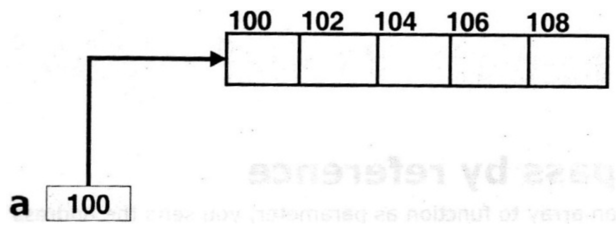
As x contains address of first element, it can be used as pointer to access the value of first element as shown below.

a = *x;

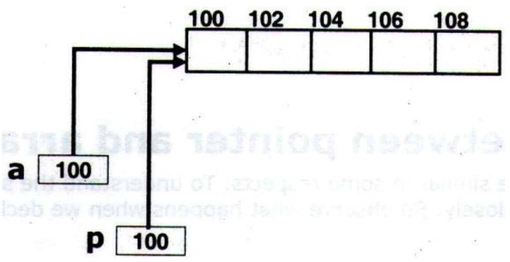
The above statement stores the value of first element of array x into variable x. This is same as a = x[0];

So an array name can be used as a pointer and the reverse is also true. The following code will use an array as a pointer and a pointer to access an array.

```
int *p;
int a[5],x;
p = a; /*p now points to first element of a*/
*p = 20; /*stores value 20 in first element of array a[0]*/
x = *a; /*will store value of first element of the array into x*/
```



An array in memory



Pointer pointing to an array

All the following will store value 20 into 0th element of the array.

*p = 20;   p[0] = 20;   *a = 20;   a[0] = 20;

Pointer Arithmetic:

All that an array name contains is the starting address of array. For the program to access the complete array just by using starting address, we need to understand pointer arithmetic.

Adding an integer to pointer

When you add an integer to pointer then the value (address) of pointer is incremented by the size of data type to which pointer points. See the following example:

```
int *p;
int a[5];
p=a; /*p points to first element of the array*/
p++; /*p now points to second element a[1]*/
```

p is a pointer to int. So when you increment p by one it is incremented by number of bytes occupied by int, which is 2 bytes. If the value of p is 1000 then when you increment p, it becomes 1002. If a float pointer (float *fp) is increment by one, then it is incremented by 4 bytes, as float occupies 4 bytes.

Subtracting an integer from pointer

Subtracting a pointer makes pointer pointing to previous element. That means the pointer is decremented by the size of the data type to which the pointer is pointing.

Example: Program to access an array using pointer

```
#include<stdio.h>
main()
{
      int *p,i;
      int a[5];
      clrscr();
      p=&a;
      printf("Enter 5 elements: ");
      for(i=0;i<5;i++) {
            scanf("%d",p);
            p++;
      }
      p=a;
      printf("The entered 5 elements: ");
      for(i=0;i<5;i++) {
           printf("%d ",*p);
           p++;
      }
      printf("\nDisplay array in reverse order: ");
      for(--p;p>=a;p--)
           printf("%d ",*p);
      getch();
}
```

Output:

```
Enter 5 elements: 4 5 2 6 3
The entered 5 elements: 4 5 2 6 3
Display array in reverse order: 3 6 2 5 4
```

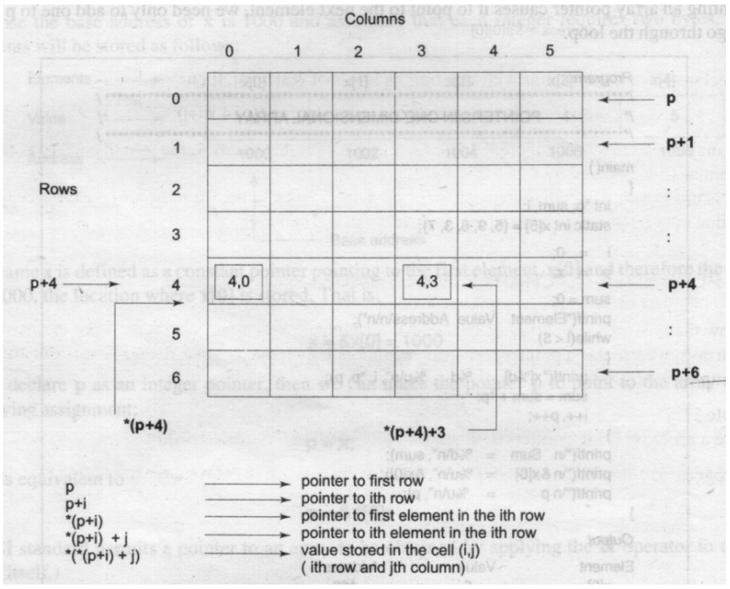
Pointers to two-dimensional arrays

Pointers can be used to manipulate two dimensional arrays as well. In one-dimensional array x, the expression

***(x+i) or *(p+i)**

represents the element x[i]. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

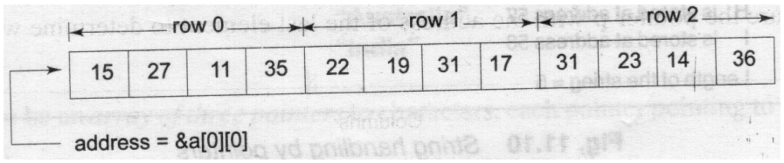
***(*(a+i)+j) or *(*(p+i)+j)**


Columns
0 1 2 3 4 5
Rows
0 1 2 3 4 5 6
p
p+1
p+4
p+6
4,0
4,3
*(p+4)
*(p+4)+3
p → pointer to first row
p+i → pointer to ith row
*(p+i) → pointer to first element in the ith row
*(p+i) + j → pointer to ith element in the ith row
((p+i) + j) → value stored in the cell (i,j) (ith row and jth column)

The figure illustrates how this expression represents the element a[i][j]. The base address of the array a is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements, row-wise. That is, the first element of the second row is placed immediately after the last element of the first row and so on. Suppose we declare an array a as follows:

int a[3][4] = {{15,27,11,35},{22,19,31,17},{31,23,14,36}};

The elements of a will be stored as shown below:



If we declare p as an int pointer with the initial address of &a[0][0], then

a[i][j] is equivalent to *(p+column_size x i+j) => *(p+4*i+j)

You may notice that, if we increment i by 1, the p is incremented by 4, the size of each row, making p element a[2][3] is given by *(p+2*4+3) = *(p+11).

That is the reason why, when a two-dimensional array is declared, we must specify the size of each row.

Example: Program to illustrate pointer to two-dimensional array

```
#include<stdio.h>
#include<conio.h>
main()
{
     int i,j,*p;
     int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
     clrscr();
     printf("Elements of An Array with their addresses\n\n");
     p=&a[0][0];
     for(i=0;i<3;i++) {
           for(j=0;j<3;j++)
                printf("[%5u]-%d ",(p+3*i+j),*(p+3*i+j));
           printf("\n");
     }
     printf("\nElements of array:\n\n");
     for(i=0;i<3;i++) {
           for(j=0;j<3;j++)
                printf("%d ",*(*(a+i)+j));
           printf("\n");
     }
     getch();
}
```

Output:

```
Elements of An Array with their addresses

[65482]-1 [65484]-2 [65486]-3
[65488]-4 [65490]-5 [65492]-6
```

```
[65494]-7 [65496]-8 [65498]-9

Elements of array:

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

Dissimilarities between pointer and array name

We used till now a pointer and array name interchangeably. But they are not identical in all respects. The following are the differences between array name and a pointer:

- ❑ First array name is constant and a pointer is a variable. You can change the value of pointer by storing an address into it. But you cannot change the address stored in array name. Because, if the address of the first element is lost then the total array will be inaccessible.
- ❑ A pointer is not initialized, whereas an array name is initialized to the address of the first element of the array.

POINTERS AND CHARACTER STRINGS

A String is an array of characters, terminated with a null character. Like in 1-D arrays, we can use pointer to access the individual characters in a string.

Example: Program to read string from keyboard and display it using character pointer.

```
#include<stdio.h>
#include<conio.h>
main()
{
       char name[15],*ch;
       printf ("Enter Your Name:");
       gets(name);
       ch=name; /* store base address of string name */
       while (*ch!='\0') {
              printf ("%c",*ch);
              ch++;
       }
       getch();
}
```

Output:

```
Enter Your Name: KUMAR

KUMAR
```

Example: Program to find length of a given string including and excluding spaces using pointers.

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

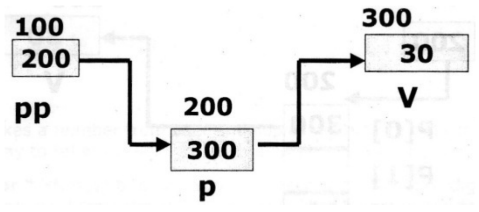
```
main()
{
     char str[20],*s;
     int p=0,q=0;
     clrscr();
     printf ("Enter String:");
     gets(str);
     s=str;
     while (*s!='\0') {
       printf ("%c",*s);
       p++; s++;
       if (*s==32) /* ASCII equivalnet of ' ' (space)is 32*/
       q++;
     }
     printf ("\nLength of String including spaces: %d",p);
     printf ("\nLength of String excluding spaces: %d",p-q);
}
```

```
OUTPUT:
Enter String: POINTERS ARE EASY

POINTERSAREEASY
Length of String including spaces: 17
Length of String excluding spaces: 15
```

POINTER to POINTER

It is possible to make a pointer pointing to another pointer. Then it becomes a pointer pointing to another pointer, which in turn points to a value. The following example, will use a pointer to a pointer to an integer.

```
int **pp; /*pointer to pointer to integer*/
int *p; /*pointer to integer*/
int v=30; /*integer*/
p=&v; /*make p pointing to v*/
pp=&p; /*make pp pointing to p*/
```



```
/*now all the below will display 30*/
printf("Value: %d",v);
printf("Value: %d",*p);
printf("Value: %d",**pp);
```

When you use **pp the following step will be taken

- First the value of pp is taken. According to figure it is 200.
- Then the value from memory location 200 is taken, which is 300 in the example.
- Then from that location, the value is taken which is 30.

Array of Pointers

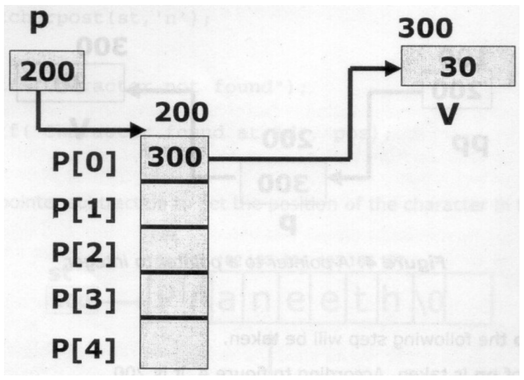
It is possible to have an array of pointers. In this case each element of the array will be a pointer.

The following code will create an array of 5 integer pointers and then make the first element pointing to an integer variable.

```
int *p[5]; int v=30;
p[0]=&v;
printf("%d",v);
printf(" %d",*p[0]);
```

Each element of the array is a pointer that can point to an integer. But remember, unless initialized all elements of the array will be uninitialized pointer. So make sure you use elements only after making them point to a valid location.

The following figure shows how an array of pointer looks like in memory.



Example: Program to display strings using array of pointers.

```
#include<stdio.h>
main()
{
      char st[10][20],*p[10];
      int i;
      clrscr();
      printf("Enter 10 strings: ");
      for(i=0;i<10;i++)
           gets(st[i]);
      printf("\nDisplay 10 strings: ");
      for(i=0;i<10;i++) {
           p[i]=&st[i];
           puts(p[i]);
      }
      getch();
}
```

WEEK - 10	
1	Structures: Definition, Syntax, Nested Structures
2	Pointers to Structures
3	Unions: Definition, Syntax

Structures

We often have to store a collection of values. When the collection is of the same type then **array** is used to store values. When the collection of values is not of the same type then a **structure** is used to store the collection.

An Array is a group of related data items or collection of homogeneous data that share a common name i.e. it contains elements of same datatype. If we want to represent a collection of data items of different types using a single name, then we cannot use an array.

C supports a constructed data type known as **structure** which is a method of packing data of different types. A **structure** is a convenient tool for handling a group of logically related data items. It is a collection of heterogeneous data that share a common name.

A **structure** is a collection of items which may be of different types referenced commonly using one name. Each item in the structure is called as a **member**.

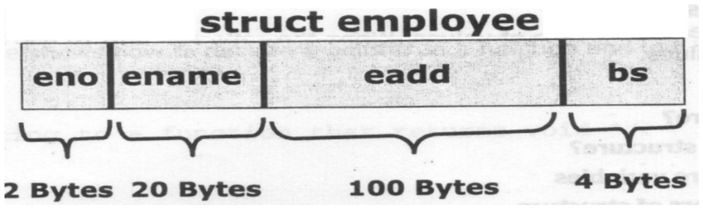
Declaration of Structure

A structure is also called **user-defined** data type. When a structure is declared a new data type is created. You declare a structure using keyword **struct**. The following is the syntax to declare a structure:

```
struct [structure-name]
{
      members declaration
}[variables];
```

If we want to store details of an employee we have to create a structure:

```
struct employee
{
      int eno;
      char ename[20];
      char eadd[100];
      float bs;
};
```



Memory Image for structure employee

The above is a structure declaration for employee structure. Employee structure contains members eno, ename, eadd, and bs. Each member may belong to a different type of data. When you declare a structure you just create a new data type - struct employee.

Structure name is otherwise known as tag. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above declaration has not declared any variables. It simply describes a format called template to represent information as shown below:

eno	Integer
ename	20 Characters
eadd	100 Characters
bs	Float

While declaring a structure you can also declare variables by giving list of variables after right brace. Members of a structure may be of standard data type or may be of another structure type.

```
struct point
{
     int x,y;
}fp,sp; /*sp and fp are two structure variables*/
```

Declaring Structure variable

You can declare structure variables using **struct structname.** You can use a variable of any structure anywhere in the program where the structure is visible. Normally structures are declared at the beginning of the program and outside all functions, so that they are global and available to the entire program.

Following is the declaration of structure variable e1:

```
struct employee e1;
```

When the variables are created to the template then the memory is allocated. The number of bytes occupied by structure variable is equal to the total number of bytes required for all members of the structure. For example, a variable of struct employee will occupy 126 bytes (2+20+100+4). So, 126 bytes are allocated for the employee structure variable e1.

Accessing a member

A structure is a collection of members. All members are to be accessed using a single structure variable. So to access each member of the structure you have to use **member operator (.)**.

We can assign values to members of a structure in a number of ways. The members themselves are not variables. They should be linked to structure variables in order to make them meaningful members.

`structure-variable.member`

The following is an example for accessing member of a structure variable.

```
struct employee x;
x.eno=10; /*place 10 into eno number*/
scanf("%f",&x.bs);
strcpy(x.ename,"Santosh");
printf("%d %s %f",x.eno,x.ename,x.bs);
```

Once you use member operator to access a member of a structure, it is equivalent to a variable of the member type. That means x.eno in the above example is equal to any integer variable as eno is of integer type.

Structure Initialization

We can initialize a structure variable by listing out values to be stored in members of the structure within braces after structure variable, while declaring the variable.

```
main()
{
struct st_record
{
      int rno;
      char name[20];
      int weight;
      float height;
};
struct st_record st1={1, "Ashok",60,180.75};
struct st_record st2={2, "Balu",50,185.72};
…
…
}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Example: Program to demonstrate how to use structure

```
main()
{
     struct employee
     {
          int eno;
          char ename[20];
          char eadd[100];
          float bs;
     };
     struct employee x;
     int hra;
     printf("Enter employee details: ");
     scanf("%d",&x.eno);
     gets(x.ename);
     gets(x.eadd);
     scanf("%f",&x.bs);

     if(x.bs>15000)
          hra = 1500;
     else
          hra = 1000;

     printf("Details of employee\n");
     printf("Employee Number: %d\n",x.eno);
     printf("Employee Name: %s\n",x.ename);
     printf("Employee Address: %s\n",x.eadd);
     printf("Employee Basic Salary: %f\n",x.bs);
     printf("Employee Salary: %f\n",x.bs+hra);
}
```

Array of structures

Just like how we create an array of standard data types like int, float etc, you can create an array of structures also. An array of structures is an array where each element is a structure. You can declare an array of structures as follows:

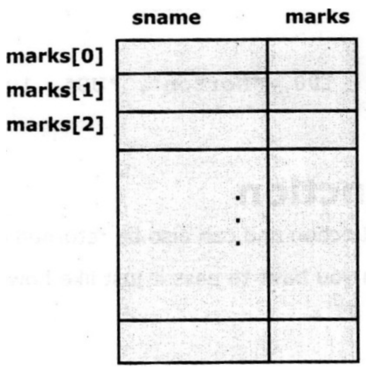
`struct employee edet[10];`

Now edet is an array of 10 elements where each element can store data related to an employee. To access eno of 5th element you would enter:

`edet[5].eno=105;`

Example: Program to take marks details of 10 students and display the name of the student with highest marks.

```
main()
{
     struct smarks
     {
          char sname[20];
          int marks;
     };
     struct smarks marks[10];
     int i,pos;
     for(i-0;i<10;i++)
     {
     printf("Enter student [%d] name: ",i);
          gets(marks[i].sname);
          printf("Enter student [%d] marks: ",i);
          scanf("%d",&marks[i].marks);
     }
     pos=0;
     for(i=1;i<10;i++)
     {
          if(marks[i].marks > marks[pos].marks)
               pos=i;
     }
     printf("Student %s has got highest marks %d\n",
                              marks[pos].sname,marks[pos].marks);
}
```



An array of smarks structure

Structures and assignment

We can copy the value of one structure variable to another when both the structure variables belong to same structure type. For instance if you have two variables called x and y of structure employee then you can copy value of y to x using simple assignment operator.

x=y;

Copying one structure variable to another is equivalent to copying all members of one to another. Interesting fact about structure assignment is even members that are of string type are copied. Remember copying a string to another is not possible without using strcpy function. But when you copy one structure variable to another of the same type, the data (including strings) will be automatically copied.

Assume that x and y are variables of employee structure, copying y to x would copy even ename and eadd though they are strings. Because copying a structure variable to another structure variable is equivalent to copying the entire block (allocated to y) to another (block allocated to x) in memory.

Pointer to Structure

A pointer can also point to structure just like how it can point to a standard data type. The process is similar to what we have already seen with standard data types.

struct employee *p; /*a pointer pointing to struct employee*/

Now p can be made to point to e1 (a variable of struct employee) as follows:

```
struct employee *p;
struct employee e1 = {1, "Srikanth","Vizag",65000};
p=&e1; /* make p point to e1 */
```

To access a member of a structure variable using a pointer to structure, use **arrow operators**. Arrow operator is combination of hyphen and greater than symbol (**->**).

pointer-to-struct -> member

In order to use p, which is a pointer to struct, to access the data of e1, which is a struct employee variable, use arrow operator as follows:

```
printf("%d  %f",  p->eno, p->bs);
```

In fact you can also use * operator to access a member through pointer. The following is the expression to access member x using pointer p.

x = (*p).x;

Remember parantheses around *p are required as without that C would take member operator (.) first and then indirection operator (*), as **.** has higher precedence than *****. That means *p.x would be taken as ***(p.x)** by C and not as **(*p).x.**

Example: Program to illustrate pointer to structure variables concept.

```
#include<stdio.h>
main()
{
      struct employee
      {
            int eno;
            char ename[20];
            float bs;
      };
```

```
	struct employee e,*emp;
	clrscr();
	printf("Enter employee name: ");
	gets(e.ename);
	printf("Enter employee number: ");
	scanf("%d",&e.eno);
	printf("Enter employee salary: ");
	scanf("%f",&e.bs);
	emp = &e;
	printf("\nEmployee Details:\n");
	printf("%d\t%s\t%0.2f",emp->eno,emp->ename,emp->bs);
	getch();
}
```

Output:

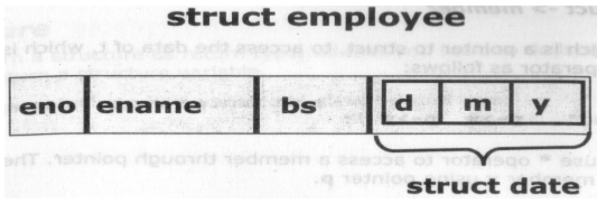
```
Enter employee name: BalaGuru
Enter employee number: 201
Enter employee salary: 27000

Employee Details:
201     BalaGuru        27000.00
```

Nested Structure

A structure may contain a member that is of another structure type. When a structure is used in another structure, it is called as **nested structure**.

Let us add a new member – date of joining (dj) to **employee** structure. DJ is consisting of members' day, month and year. So it is another structure. Here is the complete declaration of both the structures.

```
struct date
	{
		int d;
		char m[15];
		int year;
	};
struct employee
	{
		int eno;
		char ename[20];
		float bs;
		struct date dj;
	};
```



The figure above depicts the memory image of struct employee. It contains four members where member DJ again contains another three members.

Accessing nested structure

To access members of nested structure, first we need to access the member in the outer structure and then the member of nested structure. For example, the following example shows how to initialize and access nested members.

```
struct employee e1 = {1,"Ken",10000,{10,"October",2012}};

printf("Date : %d-%s-%d",e1.dj.d,e1.dj.m,e1.dj.y);
```

Example: Program to illustrate structure within structure concept.

```
#include<stdio.h>
main()
{
      struct student
      {
            int rno;
            char name[20];
            int marks;
            struct dob
            {
                  int d;
                  char mon[20];
                  int y;
            }db;
      }s[2];
      int i;
      clrscr();
      for(i=0;i<2;i++)
      {
            printf("Enter Student [%d] Roll Number: ",i+1);
            scanf("%d",&s[i].rno);
            fflush(stdin);
            printf("Enter Student [%d] Name: ",i+1);
            gets(s[i].name);
            printf("Enter Student [%d] Marks: ",i+1);
            scanf("%d",&s[i].marks);
            printf("Enter Student [%d] Date of Birth: ",i+1);
            scanf("%d %s %d",&s[i].db.d,s[i].db.mon,&s[i].db.y);
      }
      printf("\nStudents Details:\n");
      for(i=0;i<2;i++)
      {
            printf("\n%d\t%s\t%d\t%d-%s-%d\n",s[i].rno,s[i].name,
                         s[i].marks,s[i].db.d,s[i].db.mon,s[i].db.y);
```

```
    }
    getch();
}
```

Unions

Unions are the concept borrowed from structures and follow the same syntax as structures. However there is a distinction between them in terms of storage. In structures each member has its own storage location, where as all the members of union use the same location i.e. all through a union may contain many members of different types; it can handle only one member at a time. In a union, memory is allocated to only the largest member of the union. All the members of the union will share the same area. That is why only one member can be active at a time.

Like structure, a union can be declared using keyword **union** as follows:

```
union item
    {
        int m;
        char c;
        float x;
    }code;
```

This declares a variable code of type union item. The union contains 3 members, each with different data type. However we can use one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

1000	1001	1002	1003

←c→

←------m-----→

←------------------------x------------------------→

The compiler allocated a piece of storage that is large enough to hold largest variable type in the union. In the declaration above, the member x requires 4 bytes which is largest among the members.

For example, if we use x in the above example, then we should not use m and c. If you try to use both at the same time, though C doesn't stop; you overwrite one data with another. If you access m after x and c, then the value available in union will be the value of m. If you display all the members the value of m will be the same and rest all will be garbage values.

A union is used for 2 purposes:

1) To conserve memory by using the same area for two or more different variables.
2) To represent the same area of the memory in different ways.

Example: Program to illustrate concept of unions.

```
#include<stdio.h>
#include<conio.h>
main()
{
      union number
      {
            int  n1;
            float n2;
            char name[20];
      };
      union number x;
      clrscr();
      printf("\nEnter Name: ");
      gets(x.name);
      printf("Enter the value of n2: ");
      scanf("%f", &x.n2);
      printf("Enter the value of n1: ");
      scanf("%d", &x.n1);
      printf("\n\nValue of n1 = %d",x.n1);
      printf("\nValue of n2 = %f",x.n2);
      printf("\nName = %s",x.name);
      getch();
}
```

Output:

```
Enter Name: Zaheer
Enter the value of n2: 3.23
Enter the value of n1: 6

Value of n1 = 6
Value of n2 = 3.218751
Name = ♠
```

WEEK - 11	
1	Functions: Definition, Syntax, Terminology
2	Function Declaration, Classification (Arguments and Return Type)
3	Storage Classes, Sample C Programs

Functions

Functions are building blocks of a C program. Understanding functions is one of the most important steps in C language.

Definition

A **function** is a self-contained program segment that carries out a specific, well-defined task. A function is a collection of instructions that performs a specific task. Every function is given a name. The name of the function is used to invoke (call) the function. A function may also take parameters (**arguments**). If a function takes parameters, parameters are to be passed within parentheses at the time of invoking function.

A function theoretically also returns a value, but you can ignore the return value of the function in C language. Function name must follow the same rules of formation as other variables in C. The argument list contains valid variable names separated by commas.

C is a **function-oriented** language. Many operations in C language are done through functions. For example, we have used printf() function to display values, scanf() function to read values from keyboard, strlen() function to get the length of the string etc.

Classification of Functions

C functions can be classified into two categories namely

- Library functions
- User defined functions.

The main difference between these 2 categories is that

- **library functions** are not required to be written by the user, printf, scanf, strlen, sqrt and pow, etc belong to the category of library functions.
- **user defined function** has to be developed by the user at the time of writing a program. main() function is an example of user-defined function. Every program must have a main function to indicate where the program has to begin its execution.

Standard/Library functions

A function which is made available to programmer by compiler is called as standard or pre-defined function. Every C compiler provides good number of functions. All that a programmer has to do is use them straight away. For example, if you have to find length of a string , use strlen() without having to write the required code.

The code for all standard functions is available in library files, like cs.lib, and graphics.lib. These library files are supplied by the vendor of compiler. Where these libraries are stored in the system depends on the compiler. For example, if you are using Turbo C, you find libraries in LIB directory under directory where Turbo C is installed.

Declarations about standard functions are available in header files such as stdio.h, and string.h. Turbo C provides around 400 functions covering various areas like Screen IO, graphics, disk IO etc.

User-defined functions

User-defined function is a function that is defined by user. That means, the code for the function is written by user (programmer). User-defined functions are similar to standard functions in the way you call. The only difference is instead of C language providing the function, programmer creates the function.

If a program is divided into functional parts than each part may be independently coded and later combined into single unit.

Advantages of Functions:

- ❑ The length of the source program can be reduced by using functions at appropriate places.
- ❑ It becomes easier to locate a faculty functions.
- ❑ Many other programs may use a function.

User-defined functions are used mainly for two purposes:

(1) To avoid repetition of code

In programming you often come across the need to execute the same code in different places in the program. For example, if you have to print an array at the beginning of the program and at the end of the program. Then you need to write the code to display the array twice.

If you create a function to display the array, then you have to call the function once at the beginning of the program. That means, the source code need not be written for multiple times. It is written only for once and called for multiple times.

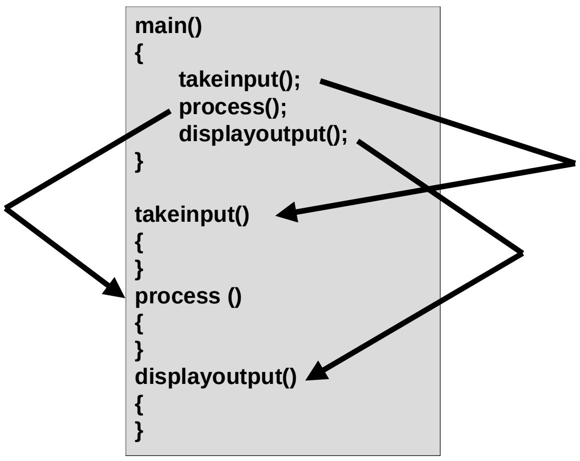
(2) To break large program into smaller units

It is never a good idea to have a single large code block. If you write entire C program as one block, the entire blocks ends up in main() function. It becomes very difficult to understand and manage.

If you can break a large code block into multiple smaller blocks, called functions, then it is much easier to manage the program. In the following example, instead of taking input, processing and displaying output in main() function, if you can divide it into three separate functions, it will be much easier to understand and manage.

```
main()
{

     /* take input here */
            . . .

     /* process the input here */
            . . .

     /* display output here */
             . . .

}
```

Single large main() function



```
main()
{
     takeinput();
     process();
     displayoutput();
}

takeinput()
{
}
process ()
{
}
displayoutput()
{
}
```

Main() function calling other functions

Creating user-defined function

A user-defined function is identical to main() function in creation. However, there are two differences between main function and a user-defined function.

- ❑ Name of main() function is standard, whereas a user-defined function can have any name.
- ❑ main() function is automatically called when you run the program, whereas a user-defined function is to be explicitly called.

Terminology of Functions

Function Declaration:

A function may contain declaration and definition. **Function declaration** specifies function name, return type, and type of parameters. This is normally given at the beginning of the program. Though it is not mandatory in all cases, it is better to declare each function. In fact header files (*.h) contain declarations of all standard functions.

The following is the syntax for function declaration.

return-type functionname (parameters);

Function declaration is called as **Prototype declaration.** Though it is not necessary in majority of cases, it is needed in cases where the following conditions apply:

- Call to function comes before definition of the function
- Function returns non-integer value.

Function Definition:

Function definition is where the statements to be executed are given. When the function is called the statements given here are executed.

```
        return-type functionname ( parameters )
        {
                statements;
                return  value;
        }
```

Return type	specifies the type of value function returns. If function doesn't return any value then it must be **void**.
Parameters	are formal parameters of the function.
Statements	are the statements to be executed when function is called.
Return	statement is used to return a value from function.

Caller and Callee

The function which calls another function is called **caller**. The function which is called by the caller is called **callee**.

Passing Parameters:

A parameter or argument is a value passed to function so that function can use that value while performing task. A function may take none, one or more parameters depending upon the need.

Example: line(); /*function that pass no parameters*/
line(20); /*function that pass one parameter*/
getaverage(10,20); /*function that pass more than one parameter*/

Actual Parameter: It is the value that is passed to function while calling the function. For example; 10 and 20 values that we passed to line() and getaverage() function in the above are actual parameters.

Formal Parameter: Variable that is used to receive actual parameter is called formal parameter.

Example: getaverage(int a, int b); Here a and b are formal parameters.

Returning Value:

Normally after performing the task functions return a value. The return value may be of any type. But a function can return only one value. To return a value from the function, we have to first specify what type of value the function returns and then we have to use return statement in the code of the function to return the value.

When return type is not explicitly mentioned it defaults to **int**. If a function doesn't return any value then specify return type as **void**.

A function returns the value to the location from where it has been called.

Storage Classes

Properties of a variable:

A variable in C language is associated with the following properties:

Property	Meaning
Name	Name is used to refer to variable. This is a symbol using which the value of the variable is accessed or modified.
Data type	Specifies what type of value the variable can store
Value	The value that is stored in the variable. The value is not known unless the variable is explicitly initialized.
Address	The address of the memory location where the variable is allocated space
Scope	Specifies the region of the program from where variable can be accessed.
Visibility	Specifies the area of the program where variable is visible.
Extent	Specifies how long variable is allocated space in memory.

Blocks:

C is a block structured language. Blocks are delimited by { and }. Every block can have its own local variables. Blocks can be defined wherever a C statement could be used. No semi-colon is required after the closing brace of a block.

Scope

The area of the program from where a variable can be accessed is called as scope of the variable. The scope of a variable determines over what parts of the program a variable is actually available for use (active). Scope of the variable depends on where the variable is declared. The following code illustrate the scope of a variable:

```
int g; /* scope of g is from this point to end of program */
main()
{
      int x; /* scope of x is throughout main function */
      . . . .
}
void sum()
{
      int y; /* scope of y is throughout function sum */
      . . . .
}
```

The scope of a variable may be either **global** or **local**.

Global Scope: It means the variable is available throughout the program. It is available to all functions that are defined after the declaration of the variable. This is the scope of the variables, which are declared outside of all functions. Variables that have global scope are called as **Global Variables**.

In the above example, variable **g** has global scope; that means the variable is accessible to all the functions of the program (main, sum) that are defined after the variable is defined.
Local Scope: Variables declared at the beginning of the block are available only to that block in which they are declared. When the scope of the variable is confined to block it is called as Local Scope. Variables with local scope are called as **Local Variables**.

In the above example, variables **x** and **y** have local scope; that means the variables are accessible only to the functions at which they are declared.

C allows you to create variables not only at the beginning of a function, you can also declare variables at the beginning of the block.

```
main()
{
     int x; /* scope of x is throughout main function */
     . . . .
     if( ...) {
          int p; /* scope of p is accessible only within if block */
          . . . .
     }
     else {
          int q; /*scope of q is accessible only within else block*/
          . . . .
     }
}
```

You cannot change the scope of a variable, it is completely dependent on the place of declaration in the program.

Visibility:
Normally visibility and scope of variables are same. In other words, a variable is visible throughout its scope.

Visibility means the region of the program in which a variable is visible. A variable can never be visible outside the scope. But it may be invisible inside the scope.

```
int g; /* scope of g is from this point to end of program */
main()
{
     int x; /* scope of x is throughout main function */
     . . . .
}
```

```
void sum()
{
      int y; /* scope of y is throughout function sum */
      int g;
      g=20; /* local variable g is used not global variable g */
}
```

In the above code, variable g has global scope. But it becomes invisible throughout the function sum as a variable with the same name is declared in that function.

When a global variable is re-declared in a function then the local variable takes precedence i.e., **g** in function sum will reference **g** in function but not **g** declared as global variable. So, in this case though **g** has scope throughout program, it is not visible throughout function sum.

For global variable to get its visibility in a function, when a local variable with same variable name is declared and use we use **scope resolution operator(::)**. This helps in accessing global variable even though another variable with same name is existing.

```
int g; /* scope of g is from this point to end of program */
main()
{
      int x; /* scope of x is throughout main function */
      . . . .
}
void sum()
{
      int y; /* scope of y is throughout function sum */
      int g;
      g=20; /* local variable g is used not global variable g */
      ::g=g+10; /*global variable g is used now with the operator*/
          /*'::' called scope resolution operator*/
}
```

Extent/Lifetime

Also called as **Longevity**. This refers to the period of time during which a variable resides in the memory and retains a given value during the execution of a program. Longevity has a direct effect on the utility of a given variable.

A variable declared inside a block has **local extent**. Because it exists in the memory as long as the block is in execution. It is created when block is invoked and removed when the execution of the block is completed.

On the other hand, variable declared outside all functions has **static extent** as they remain in memory throughout the execution of program.

Though variable is available in memory, we can access it only from the region of the program to which the variable is visible. That means, just because the variable is there in the memory we cannot access it.

STORAGE CLASS SPECIFIERS

'Storage' refers to the scope of a variable and memory allocated by compiler to store that variable. Scope of a variable is the boundary within which a variable can be used. Storage class defines the scope and lifetime of a variable.

From the point view of C compiler, a variable name identifies physical location from a computer where variable is stored. There are two memory locations in a computer system where variables are stored as: Memory and CPU Registers.

Functions of storage class:

- ❑ To determine the location of a variable where it is stored?
- ❑ Set initial value of a variable or if not specified then setting it to default value.
- ❑ Defining scope of a variable.
- ❑ To determine the life of a variable.

Syntax:

[storage-class] data type variable [= value] ;

Types of Storage Classes

- auto
- register
- static
- extern

auto or Automatic Storage class

This is normally not used as it is always implicitly associated with all local variables. The keyword used for this storage class is **auto**. An auto variable is automatically created and initialized when control enters into a block and removed when control comes out of block.

```
auto int i=30; /*an auto variable with local scope and extent*/
```

Storage Location: Main Memory
Default Value: Garbage
Scope: Local to the block in which variable is declared
Lifetime: till the control remains within the block.

Example: Program to illustrate how auto variables work.

```
main() {
       auto int i=10;
       clrscr();
```

```
	{
		auto int i=20;
		printf("%d",i);
	}
	printf("\t%d",i);
}
```

```
Output:
20    10
```

register or Register Storage class

We can tell the compiler that a variable should be kept in **registers** (which are memory locations on microprocessors used for internal operations), instead of keeping in the memory (where normal variables are stored).

Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g. loop control variables) in the registers will lead to faster execution of program. Most of the compilers allow only int or char variables to be placed in the register.

```
register int i;
```

The above is a request to the compiler to allocate a register to variable **i** instead of allocating memory. If the register is available then **i** is placed in register, otherwise it is as usual given memory location.

The programmer doesn't know whether register is allocated or memory is allocated to a variable that is used with register storage class. So, it is not possible to use address operator (&) with variables that contain register storage specifier. This is of the fact that registers do not contain address.

It is not applicable for arrays, structures or pointers. It cannot not used with static or external storage class.

Storage Location: CPU registers
Default Value: Garbage
Scope: Local to the block in which variable is declared
Lifetime: till the control remains within the block.

Since, there are limited number of register in processor and if it couldn't store the variable in register, it will automatically store it in memory.

Example:

```
main()
{
	register int i;
	for(i=0;i<10000;i++)
		. . . .
}
```

static or Static Storage Class

static is the default storage class for global variables. This is used to promote local extent of a local variable to static extent.

The value of static variables persists until the end of the program. A variable can be declared static using the keyword **static** like

```
      static int x;
      static float y;
```

When you use static storage class while declaring local variable, instead of creating the variable and initializing it whenever control enters the block, the variable is created and initialized only for once - when variable's declaration is encountered for first time. So a variable with static extent will remain in the memory across function calls.

Storage Location: Main Memory
Default Value: Zero
Scope: Local to the block in which variable is declared
Lifetime: till the value of the variable persists between different function calls.
Example: Program to illustrate the properties of a static variable.

```
main() {
       int i;
       for (i=0; i<3; i++)
              incre();
}
incre() {
       int avar=1;
       static int svar=1;
       avar++; svar++;
       printf("\n Automatic variable value : %d",avar);
       printf("\t Static variable value : %d",svar);
}
```

Output:

```
Automatic variable value: 2       Static variable value: 2
Automatic variable value: 2       Static variable value: 3
Automatic variable value: 2       Static variable value: 4
```

extern or External Storage class:

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function.
In case of large program, containing more than one file, if the global variable is declared in file 1 and that variable is used in file 2 then, compiler will show error. To solve this problem, keyword **extern** is used in file 2 to indicate that, the variable specified is global variable and declared in another file.

```
/* File1.c */
extern int count; /* refers to an external variable count */
inccount() {
       count++;
}

/* File2.c */
#include"File1.c"
int count; /* creates variable count with static extent */
main() {
       . . .
}
```

In this case program File1.c has to access the variable defined in program File2.c.

Storage Location: Main Memory

Default Value: Zero

Scope: Global/File Scope

Lifetime: as long as the program execution doesn't come to an end.

Example: Program to illustrate extern storage class

write.c

```
void write_extern();
extern int count;
void write_extern()
{
       count = count + 2;
       printf("Count is %d\n",count);
}
```

mmain.c

```
#include"write.c"
int count=5;
main() {
       write_extern();
       write_extern();
}
```

```
Output:
Count is 7
Count is 9
```

Category of Functions:

A function depending on whether arguments are present or not and whether a value is returned or not may belong to one of the following categories:

1. Functions with no arguments and no return value.
2. Functions with arguments and no return value.
3. Functions with arguments and return value.
4. Functions with no arguments and return value.

Functions with no arguments and no return value

When a function has no argument, it does not receive any data from calling function. When it does not return a value, the calling function does not receive any data from called function. In effect, there is no data transfer between calling function & called function.

Program 1:

```
#include<stdio.h>
main()
{
	printf("Text in main Function\n");
	print();
}
print()
{
	printf("\nText in print function");
}
```

Program 2:

```
#include<stdio.h>
main()
{
	printf("Addition of 2 numbers:\n");
	sum();
}
sum()
{
	int a,b;
	printf("Enter 2 numbers: ");
	scanf("%d %d",&a,&b);
	printf("\nSum of %d and %d is %d",a,b,a+b);
}
```

Functions with no arguments and return value

When a function has no argument, it does not receive any data from calling function. When it does return a value, the calling function receives data from called function. In effect, there is data transfer between called function & calling function.

Program:

```
#include<stdio.h>
float average();
main()
{
	float avg;
	printf("Average of 3 numbers:\n");
	avg = average();
	printf("Average: %f",avg);
}
float average()
{
	int a,b,c,s;
	printf("Enter 3 numbers: ");
	scanf("%d %d %d",&a,&b,&c);
	s=a+b+c;
	return s/3.0;
}
```

Functions with arguments and no return value

When a function has argument(s), it does receive data from calling function. When it does not return a value, the calling function receives does not data from called function. In effect, there is data transfer between calling function & called function.

Program 1:

```
#include<stdio.h>
main()
{
	dline(15);
	printf("\nFunctions...");
	dline(15);
	getch();
}
dline(int n)
{
	int l;
	for(l=0;l<n;l++)
		putch('-');
}
```

Program 2:

```
#include<stdio.h>
void interest(float,int,float);
main()
{
	int t;
	float p,r;
	printf("Enter principal amount: ");
	scanf("%f",&p);
	printf("Enter time period: ");
	scanf("%d",&t);
	printf("Enter rate: ");
	scanf("%d",&r);
	interest(p,t,r);
}
void interest(float pr,int tp,float rate)
{
	float si;
```

```
     si = (pr*tp*rate)/100;
     printf("Simple Interest for Rs %0.2f is %0.2f",pr,si);
}
```

Functions with arguments and return value

When a function has an argument, it receives back any data from calling function. When it returns a value, the calling function receives any data from called function. In effect, there is a data transfer between the calling function & called function.

Program:

```
int getsum(int n);
main()
{
     int v,sum;
     printf("Enter n value: ");
     scanf("%d",&v);
     sum = getsum(v);
     printf("Sum=%d",sum);
}
int getsum(int n)
{
     int s=0;
     for(;n>0;n--)
          s+=n;
     return s;
}
```

WEEK - 12	
1	Parameter Passing Techniques
2	Passing Parameters Types
3	Recursion

Parameter Passing Techniques: Call by Value & Call by Reference

Call by Value:

If data is passed by value, the data is copied from the variable used in main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

When we call a function then we will just pass the variables or the arguments and we doesn't pass the address of variables , so that the function will never effects on the values or on the variables.

So Call by value is that the values those are passed to the functions will never effect the actual values those are Stored into the variables.

Example: Program to illustrate the concept of call by value

```
#include <stdio.h>
swap (int, int);
main()
{
      int a, b;
      printf("\nEnter value of a & b: ");
      scanf("%d %d", &a, &b);
      printf("\nBefore Swapping:\n");
      printf("\na = %d\n\nb = %d\n", a, b);
      swap(a, b);
      printf("\nAfter Swapping:\n");
      printf("\na = %d\n\nb = %d", a, b);
      getch();
}
swap (int a, int b)
{
      int temp;
      temp = a;
      a = b;
      b = temp;
}
```

Output:

```
Enter value of a & b: 2 3
Before swapping: 2 3
After swapping: 2 3
```

Call by Reference:

If data is passed by reference, a pointer to the data is copied instead of the actual variable as it is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main().

In the call by reference we pass the address of the variables whose arguments are also send. So that when we use the reference then, we pass the address the variables.

When we pass the address of variables to the arguments then a function may effect on the variables. So, when a function will change the values then the values of variables gets automatically changed and when a function performs some operation on the passed values, then this will also effect on the actual values.

Example: Program to illustrate the concept of call by reference

```
#include <stdio.h>
swap (int *, int *);
main()
{
      int a, b;
      printf("\nEnter value of a & b: ");
      scanf("%d %d", &a, &b);
      printf("\nBefore Swapping:\n");
      printf("\na = %d\n\nb = %d\n", a, b);
      swap(&a, &b);
      printf("\nAfter Swapping:\n");
      printf("\na = %d\n\nb = %d", a, b);
      getch();
}
swap (int *x, int *y)
{
      int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}
```

Output:

```
Enter value of a & b: 2 3
Before swapping: 2 3
After swapping: 3 2
```

Passing an array as argument to function:

An array is actually held as an address to an area of memory. This means that when you come to pass it into a function, it can only be passed in as a reference.

Also, when you define the parameter to read in the data in the function prototype, you declare it as though you were reading in a single-item variable instead of an array. When an array is passed as parameter, only the name of the array is given at the time of calling function the formal parameter is to be declared as an array.

It is then up to the function to read the first item from the array, move the pointer along to the next item, and then carry on until it finds the end of the array - thus, it has to somehow know how long the array is.

```
function_name(int x[5])
function_name(int x[])
```

The parameter x in the above declarations are also internally taken as an integer pointer by C to access the elements of the array from the first element.

Passing a 1-dimensional array as argument to function:

Example 1: Program to print the given array

```
printarray(int []);
main()
{
      int a[5],i;
      for(i=0;i<5;i++)
            scanf("%d",&a[i]);
      printarray(a);
}
printarray(int ar[5])
{
      int i;
      for(i=0;i<5;i++)
            printf("%d\n",i[ar]);
}
```

Example 2: Program to find largest from the given array

```
#define SIZE 50
int big(int [], int);
main()
{
      int a[SIZE], n, i, b;
      printf("\nEnter size of array: ");
      scanf("%d", &n);
      printf("\nEnter elements:\n");
      for (i=0; i<n; i++)
            scanf("%d", &a[i]);
      b = big(a, n);
      printf("\nLargest number: %d", b);
}
int big(int a[], int n)
{
      int b, i;
      b = a[0];
      for(i=0; i<n; i++)
            if (a[i] > b) b = a[i];
      return b;
}
```

Passing a 2-dimensional array as argument to function:

Example: Program to find Diagonal elements of given matrix

```
#define ROW 10
#define COL 10
int trace(int [][], int, int);
main()
{
	int a[ROW][COL], row, col, i, j, sum;
	printf("\nEnter no. of rows and columns of a matrix: ");
	scanf("%d %d", &row, &col);
	printf("\nEnter elements:\n");
	for (i=0; i<row; i++)
		for (j=0; j<col; j++)
			scanf("%d", &a[i][j]);
	printf("\nMatrix is:\n\n");
	for (i=0; i<row; i++)
	{
		for (j=0; j<col; j++)
			printf("\t%d", a[i][j]);
		printf("\n");
	}
	sum = trace(a, row, col);
	printf("\nSum: %d", sum);
}
int trace(int x[ROW][COL], int r, int c)
{
	int i, j, s=0;
	for (i=0; i<r; i++)
		for (j=0; j<c; j++)
			if (i == j) s = s + x[i][j];
	return s;
}
```

Passing String as Parameter to function

This is similar to the concept of passing an array to a function as argument. With strings (i.e. character arrays) - we have one advantage: we know that the end of the array is marked by a zero character, so as long as we know where the beginning of the array is, we can find the end of the string of characters by moving from the beginning of the array to the end one-character at a time.

Example: Program to modify a string and display.

```
main()
{
	char st[20];
	printf("Enter Name: ");
	gets(st);
	modifystring(st);
	printf("\n%s",st);
}
modifystring(char st[20]) {
	strcat(st," Welcome");
}
```

Passing structure as argument to function

A structure can be passed to a function. To pass a structure to a function you have to pass it just like how you would pass standard data types like int and float.

Example: Program to illustrate the concept of passing structure to function.

```
struct point
{
     int x,y;
};
void display(struct point t);
main()
{
     struct point p;
     printf("Read Coordinates of a point: ");
     scanf("%d %d",&p.x,&p.y);
     display(p);
}
void display(struct point t)
{
     printf("Coordinates of a point: x=%d, y=%d",t.x,t.y);
}
```

When you are sending a structure to a function, make sure that both calling function and called function have access to structure. This can be done if you declare the structure at the beginning of the program.

Just like how we send a structure variable to a function you can also send an array of structures to a function.

Return a Structure

A function can also return a structure as return type. All that you have to do is specify return type as structure and return a structure variable.

Example: Program to call a function that returns a variable of structure.

```
struct point {
     int x,y;
};
struct point getpoint();
main()
{
     struct point p;
     p = getpoint();
     printf("Coordinates of a point: x=%d, y=%d",p.x,p.y);
}
struct point getpoint()
{
     struct point t;
     printf("Enter point coordinates: ");
     scanf("%d %d",&t.x,&t.y);
     return t;
}
```

Passing array to function and accessing array with pointer

Whenever you send an array to function as parameter, you send the address of the array and not the complete array. That means, whenever an array is passed it is passed by reference. The formal parameter in called function is nothing but a pointer pointing to first element of array.

function_name(int *x): This declaration declares x as an integer pointer. All that C needs here is a pointer to integer.

Example: Program to display the given array of elements.

```
main()
{
      int arr[10],i;
      printf("Enter 10 elements: ");
      for(i=0;i<10;i++)
            scanf("%d",&arr[i]);
      displayarray(arr);
}
displayarray(int *a)
{
      int i;
      printf("Elements in reverse order: ");
      for(i=10-1;i>=0;i--)
            printf("%d ",*(a+i));
}
```

Passing string to function and accessing string with pointer

Strings which are made up of characters, we'll be using pointers to characters. However, pointers only hold an address; they cannot hold all the characters in a character array. This means that when we use a char * to keep track of a string, the character array containing the string must already exist.

Example: Program to accept string and display vowels

```
vowels(char *p);
main()
{
      char str[10];
      printf("\nEnter the string: ");
      gets(str);
      vowels(str);
}
vowels(char *p)
{
      int i,l;
      l=strlen(p);
      printf("\n\nThe vowels in the string are: ");
      for(i=0;i<l;i++)
      {
            if(p[i]=='a' || p[i]== 'e' || p[i]=='i' || p[i]=='o' ||
                  p[i]=='u'|| p[i]=='A' || p[i]=='E' || p[i]=='I' ||
                  p[i]=='O' || p[i]=='U')
            {
                printf("%c",p[i]);
            }
            else
                continue;
        }
}
```

Calling Function within Function

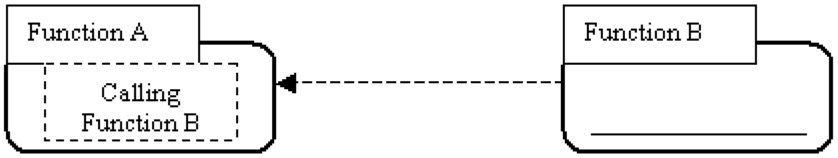
One of the main reasons of using various functions in your program is to isolate assignments; this allows you to divide the jobs among different entities so that if something is going wrong, you might easily know where the problem is.

Functions trust each other, so much that one function does not have to know HOW the other function performs its assignment. One function simply needs to know what the other function does, and what that other function needs.

Once a function has been defined, other functions can use the result of its assignment. Imagine you define two functions A and B.

Function A
Function B

If Function A needs to use the result of Function B, function A has to use the name of function B. This means that Function A has to "call" Function B:



When calling one function from another function, provide neither the return value nor the body, simply type the name of the function and its list of arguments, if any.

Example: Program to call function inside a function.

```
main()
{
        processinput();
        getch();
}
processinput()
{
        int a,b,sum;
        printf("Enter 2 numbers: ");
        scanf("%d %d",&a,&b);
        sum = process(a,b);
        displayresult(a,b,sum);
}
process(int a,int b)
{
        return a+b;
}
displayresult(int x,int y,int s)
```

```
{
     printf("Sum of 2 numbers %d and %d is %d",x,y,s);
}
```

Pointer to Function

It is also possible to have a pointer pointing to a function. A pointer to a function contains the address of the location to which control is transferred when you use the pointer.

The following example shows how to declare a pointer to a function and use it to invoke the function.

```
/*pointer pointing to function that returns void*/
void (*fp)();
void print();
main()
{
     /*make fp pointing to print function*/
     fp = print;
     /*call print function through pointer to function*/
     (*fp)();
}
void print()
{
     printf("Hello! From Function Print...");
}
```

Recursion

Recursion is a process of defining something in terms of itself and is sometimes called circular definition. When a function calls itself it is called as recursion. Recursion is natural way of writing certain functions.

Recursive Function: A function is said to be recursive if a statement in the body of the function calls itself. Each recursive function must specify an exit condition for it to terminate, otherwise it will go on indefinitely.

A simple example of recursion is presented below:

```
main()
{
     printf("\nThis is an example of recursion");
     main();
}
```

When executed, this program will produce an output like:

```
  This is an example of recursion
  This is an example of recursion
  …
  …
  …
  This is an example of recursion
  …
```

Execution must be terminated abruptly (suddenly or unexpected) or forcibly, otherwise the execution will continue infinitely.

Recursive functions can be effectively used to solve problems where the solution is expressed in terms of successively applying the same solution of the subset of the problem.

When we write recursive functions we must have an **if** statement somewhere to force the function to return without recursive calls being executed. Otherwise function will never return back and will be terminated with an error saying “out of stack space”.

Example: Program to find sum of n below numbers using recursion.

```
#include <stdio.h>
main()
{
     int n,sum;
     printf("\nEnter n value: ");
     scanf("%d", &n);
     sum = sum_num(n);
     printf("\nSum of %d below numbers is %d",n,sum);
     getch();
}
sum_num(int n)
{
     if (n == 1)
          return n;
     else
          return n+sum_num(n-1);
}
sum_num(5) returns (5 + sum_num(4),

     which returns (4 + sum_num(3),

     which returns (3 + sum_num(2),

          which returns (2 + sum_num(1),

                    which returns (1)))))
```

Example: Program to find factorial of given number using recursion.

```
#include <stdio.h>
long fact(int);
main()
{
     int n;
     long f;
     printf("\nEnter number to find factorial: ");
     scanf("%d", &n);
     f = fact(n);
     printf("\nFactorial: %ld", f);
     getch();
}
```

```
long fact(int n)
{
     int m;
     if (n == 1)
          return n;
     else
          return n * fact(n-1);
}
```

Example: Program to find the GCD of two given integers using Recursion

```
#include<stdio.h>
#include<conio.h>
int gcd (int, int); //func. declaration.
void main( )
{
    int a, b, res;
    clrscr( );
    printf("Enter the two integer values:");
    scanf("%d%d", &a, &b);
    res= gcd(a, b); // calling function.
    printf("\nGCD of %d and %d is: %d", a, b, res);
    getch( );
}
int gcd( int x, int y) //called function.
{
    int z;
    z=x%y;
    if(z==0)
    return y;
    gcd(y,z); //recursive function
}
```

Example: Program to generate the Fibonocci series using Recursion.

```
#include<stdio.h>
#include<conio.h>
int fibno(int); // function declaration.
void main()
{
    int ct, n, disp;
    clrscr( );
    printf("Enter the no. of terms:");
    scanf("%d", &n);
    printf("\n The Fibonocci series:\n");
    for( ct=0; ct<=n-1; ct++)
    {
        disp= fibno(ct); //calling function.
        printf("%5d", disp);
    }
    getch( );
}

int fibno( int n)
```

```
{
    int x, y;
    if(n==0)
    return 0;
    else if(n==1)
    return 1;
    else
    {
        x= fibno( n-1);
        y= fibno( n-2);
        return (x+y);
    }
}
```

WEEK - 13	
1	Files: Definition, Opening, Closing of Files
2	Reading and Writing of Files
3	Sample C Programs

File Handling

If the program requires the input data either 1 or 2, the user could supply the data at any time. But if the details of the student in a very large institution are required to prepare marks statement for all, we have to enter the whole details every time to our program. This requires large man power, more time & computational resources; which is a disadvantage with the traditional way of giving inputs to a program for execution with the basic I/O functions.

When the data of the students are stored permanently, the data can be referred later and makes the work easier for the user to generate the marks statement. When you have to store data permanently, we have to use **FILES**. The files are stored in disks.

DEFINITION:
A file can be defined as a collection of bytes stored on the disk under a name. A file may contain anything, in the sense the contents of files may be interpreted as a collection of records or a collection of lines or a collection of instructions etc.

A file is a collection of records. Each record provides information to the user. These files are arranged on the disk.

Basic operations on files include:

- Open a file
- Read data from file/Write data to file
- Close a file

To access the data in a file using C we use a predefined structure **FILE** present in stdio.h header file, that maintains all the information about files we create (such as pointer to char in a file, end of file, mode of file etc).

FILE is a data type which is a means to identify and specify which file you want to operate on because you may open several files simultaneously.

When a request is made for a file to be opened, what is returned is a pointer to the structure FILE. To store that pointer, a pointer variable is declared as follows:

```
FILE *file_pointer;
```
where file_pointer points to first character of the opened file.

Opening of a file:

A file needs to be opened when it is to be used for read/write operation. A file is opened by the fopen() function with two parameters in that function. These two parameters are file name and file open mode.

Syntax: `fopen(filename, file_open_mode);`

The fopen function is defined in the “stdio.h” header file. The filename parameter refers to any name of the file with an extension such as “data.txt” or “program.c” or "student.dat" and so on.

File open mode refers to the mode of opening the file. It can be opened in ‘read mode’ or ‘write mode’ or ‘append mode’.
The function fopen() returns the starting address of file when the file we are trying to open is existing (i.e. success) else it returns NULL which states the file is not existing or filename given is incorrect.

E.g.:
```
FILE *fp;
fp=fopen("data.txt","r");
/*If file exists fp points to the starting address in memory
Otherwise fp becomes NULL*/
if(fp==NULL) printf("No File exists in Directory");
```

NULL is a symbolic constant declared in stdio.h as 0.

Modes of Operation:
1. "r" (read) mode: open file for reading only.
2. "w" (write) mode: open file for writing only.
3. "a" (append) mode: open file for adding data to it.
4. "r+" open for reading and writing, start at beginning
5. "w+" open for reading and writing (overwrite file)
6. "a+" open for reading and writing (append if file exists)

When trying to open a file, one of the following things may happen:
- When the mode is ‘writing’ a file with the specified name is created if the file does not exist. The contents are deleted, if the file is already exists.
- When the purpose is ‘appending’, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
- When the purpose is ‘reading’, and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

With these additional modes of operation (mode+), we can open and use a number of files at a time. This number however depends on the system we use.

Closing a file:

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all the links to the file are broken. It also prevents any accidental misuse of file.

Closing of unwanted files might help open the required files. The I/O library supports a function to do this for us.

Syntax: `fclose(file_pointer);`
/*This would close the file associated with the FILE pointer file_pointer*/
`fcloseall();`
/*This would close all the opened files. It returns the number of files it closed. */

Example: Program to check whether given file is existing or not.

```
#include<stdio.h>
main()
{
FILE *fp;
fp=fopen("data.txt","r");
if(fp==NULL) {
           printf("No File exists in Directory");
           exit(0);
      }
      else
           printf("File Opened");
      fclose(fp);
      getch();
}
```

Reading and Writing Character on Files

To write a character to a file, the input function used is putc or fputc. Assume that a file is opened with mode 'w' with file pointer fp. Then the statement,

putc(character_variable,file_pointer);
fputc(character_variable,file_pointer);

writes the character contained in the 'character_variable' to the file associated with FILE pointer 'file_pointer'.

E.g.: putc(c,fp);
fputc(c,fp1);

To read a character from a file, the output function used is getc or fgetc. Assume that a file is opened with mode 'r' with file pointer fp. Then the statement,

character_variable = getc(file_pointer);
character_variable = fgetc(file_pointer);

read the character contained in file whose FILE pointer 'file_pointer' and stores in 'character_variable'.

E.g.: getc(fp);

fgetc(fp);

The file pointer moves by one character position for every operation of getc and putc. The getc will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

Example: Program for Writing to and reading from a file

```
#include<stdio.h>
main()
{
      FILE *f1;
      char c;
      clrscr();
      printf("Write data to file: \n");
      f1 = fopen("test.txt","w");
      while((c=getchar())!='@')
            putc(c,f1);
      /*characters are stored into file until '@' is encountered*/
      fclose(f1);
      printf("\nRead data from file: \n");
      f1 = fopen("test.txt","r"); /*reads characters from file*/
      while((c=getc(f1))!=EOF)
            printf("%c",c);
      fclose(f1);
      getch();
}
```

Example: Program to write characters A to Z into a file and read the file and print the characters in lowercase.

```
#include<stdio.h>
main()
{
      FILE *f1;
      char c;
      clrscr();
      printf("Writing characters to file... \n");
      f1 = fopen("alpha.txt","w");
      for(ch=65;ch<=90;ch++)
            fputc(ch,f1);
fclose(f1);
      printf("\nRead data from file: \n");
      f1 = fopen("alpha.txt","r");
      /*reads character by character in a file*/
      while((c=getc(f1))!=EOF)
            printf("%c",c+32); /*prints characters in lower case*/
      fclose(f1);
```

```
     getch();
}
```

Example: Program to illustrate the append mode of operation for given file.

```
#include<stdio.h>
main()
{
     FILE *f1;
     char c,fname[20];
     clrscr();
     printf("Enter filename to be appended: ");
     gets(fname);
     printf("Read data from file: \n");
     f1 = fopen(fname,"r");
     if(f1==NULL)
     {
          printf("File not found!");
          exit(1);
     }
     else
     {
          while((c=getc(f1))!=EOF)
               printf("%c",c);
     }
     fclose(f1);
     f1 = fopen(fname,"a");
     while((c=getchar())!='@')
          putc(c,f1);
     /*characters are appended into file with existing data until 
'@' is encountered*/
     fclose(f1);
     getch();
}
```

Reading and Writing Integers getw and putw functions:

These are integer-oriented functions. They are similar to getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data.

The general forms of getw and putw are:

putw: Writes an integer to a file.

```
     putw(integer,fp);
```

getw: Reads an integer from a file.

```
     getw(fp);
```

Example: Program to illustrate getw and putw functions.

```
#include<stdio.h>
main()
{
     int n,num,i,sum=0;
     FILE *fp;
     clrscr();
     printf("Enter the number of integers to be written to file: ");
     scanf("%d",&n);
     fp = fopen("numbers.txt","w");
     for(i=0;i<n;i++)
     {
          scanf("%d",&num);
          putw(num,fp);
     }
     fclose(f1);
     fp = fopen("numbers.txt","r");
     while((num=getw(fp))!=EOF)
     {
          sum=sum+num;
     }
fclose(fp);
printf("Sum of %d numbers is %d\n\n",n,sum);
printf("Average of %d numbers is %d",sum/n);
getch();
}
```

End of File: feof()

The feof() function can be used to test for the end of the file condition. It takes a FILE pointer as its only argument and returns a non-zero integer value if all the data from specified file has been read and returns zero otherwise.

This function returns true if you reached end of file in given file, otherwise returns false.

```
while(!feof(fp))
{
....
....
....
}
/*executes set of statements in the loop until EOF is encountered*/
```

fgets and fputs functions:

fputs writes string followed by new line into file associated with FILE pointer fp.

    fputs(char *st, FILE *fp);

E.g.: fputs(str,fp);

fgets reads characters from current position until new line character is encountered or len-1 bytes are read from file associated with FILE pointer fp. Places the characters read from the file in the form of a string into str.

fgets(char *st,int len,FILE *fp);

E.g.: fgets(str,50,fp);

Example: Program to illustrate fgets and fputs function.

```
#include<stdio.h>
main()
{
      FILE *fp;
      char s[80];
      clrscr();
      fp=fopen("strfile.txt","w");
      printf("\nEnter a few lines of text:\n");
      while(strlen(gets(s))>0)
      {
            fputs(s,fp);
            fputs("\n",fp);
      }
      fclose(fp);
      printf("Content in file:\n");
      fp=fopen("strfile.txt","r");
      while(!feof(fp))
      {
            puts(s);
            fgets(s,80,fp);
      }
      fclose(fp);
      getch();
}
```

fprintf and fscanf functions

The functions fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions, except of course that they work on files. The first argument of these functions is a file pointer, where specifies the file to be used.

fprintf(): Writes given values into file according to the given format. This is similar to printf but writes the output to a file instead of console.

fscanf(): Reads values from file and places values into list of arguments. Like scanf it returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

The general form of fprintf and fscanf is

```
fprintf(file_pointer/stream, “control string”, list);
fscanf(file_pointer/stream, “control string”, list);
```

where file_pointer is a pointer associated with a file that has been opened for writing.

The stream refers to a stream or sequence of bytes that can be associated with a device or a file. The following are the available standard file pointers/streams:

- **stdin** refers to standard input device, which is by default keyboard.
- **stdout** refers to standard output device, which is by default screen.
- **stdprn** refers to printer device.
- **stderr** refers to standard error device, which is by default screen.

The control string (format specifier) contains output specifications for the items in the list.

The list may include variables, constants and strings.

E.g.:
fprintf(fp, “%s %d %f”, name, age, 7.5); /*writes to file*/
fprintf(stdout, “%s %d %f”, name, age, 7.5); /*prints data on console*/

fscanf(fp, “%s %d %f”, name, &age, &per); /*reads from file*/
fscanf(stdin, “%s %d %f”, name, &age, &per); /*reads data from console*/

Example: Program to illustrate fprintf and fscanf function.

```
#include<stdio.h>
main()
{
      FILE *fp;
      int number,quantity,i;
      float price,value;
      char item[20],filename[10];
      clrscr();
      printf("Enter filename: ");
      gets(filename);
      fp=fopen(filename,"w");
      printf("Enter Inventory data: \n");
      printf("Enter Item Name, Number, Price and Quantity
(3 records):\n");
      for(i=1;i<=3;i++) {
            fscanf(stdin,"%s %d %f %d",item,&number,&price,&quantity);
            fprintf(fp,"%s %d %0.2f %d",item,number,price,quantity);
      }
      fclose(fp);
      fprintf(stdout,"\n\n");
      fp=fopen(filename,"r");
      printf("Item Name\tNumber\tPrice\tQuantity\tValue\n");
      for(i=1;i<=3;i++)
```

```
     {
          fscanf(fp,"%s %d %f %d",item,&number,&price,&quantity);
          value=price*quantity;

     fprintf(stdout,"%s\t%10d\t%8.2f\t%d\t%11.2f\n",item,number,price,
               quantity,value);
     }
     fclose(fp);
     getch();
}
```

WEEK - 14	
1	Binary Files, Random Accessing of Files
2	Enum, Typedef
3	Preprocessor Commands, Sample C Programs

BINARY FILES

Binary file is a file that is used to store any type of data. Binary files are typically used to store data records.

Text file is a file which contains readable characters and a few control characters like tab, new line etc. Example for text file is .txt, .c and etc. If the contents in a file are readable then it is a text file. Text file is terminated with a special character ^Z or any other character depending on programmer's choice.

A Binary file is a file in which anything goes. Example for binary file is .exe, .obj, .dat and .com file. In these files, the contents are unreadable and show meaningless characters. Binary files are not terminated with any special character.

As Binary files don't have termination character, we have to specify whether you are dealing with text file or binary file. This is done using **b (binary)** qualifier in the modes of operation as (**wb, rb, ab, wb+, rb+, ab+**). The modes perform the same operation as like for text files.

Data file (.dat) is example of binary file. Data file is a collection of records. Each record is a structure. Whatever you have in a structure variable in memory that is copied to file as it is. So, if an integer is occupying two bytes the same number of bytes even on the file and the exact memory image is copied to file.

Writing to binary file using fwrite()

The function fwrite can be used to write a block of memory into file. Whatever you have in that block of memory that is written into file as it is.

int fwrite(void *address, int size, int noblocks, FILE *fp);

void *address	is starting address of the block that is to be written. fopen takes bytes from this address in memory.
int size	Size of each block in bytes. Use of sizeof operator to get exact number of bytes to be written while you are dealing with structures.
int noblocks	How many blocks are to be written? The size of each block is specified by int size. So, total number of bytes written will be size*noblocks

FILE *fp	Identifies the file into which the data is to be written.

fwrite returns the number of blocks written and NOT number of bytes written.

E.g.: fwrite(&s, sizeof(s), 1, fp);

In this, one structure is written to file. The starting address of the structure variable is taken using address operator &s. Size is taken using sizeof operator.

Reading binary files with fread()

fread() is same as fwrite() in syntax but does the opposite. It reads a block from file and places that block into memory (a structure variable).

int fread(void *address, int size, int noblocks, FILE *fp);

fread returns the number of blocks read. If fread() couldn't read record successfully it returns 0.

E.g.: fread(&s, sizeof(s), 1, fp);

This place's the data read from file into structure.

Example: Program to illustrate fread() and fwrite() functions.

```
#include<stdio.h>
struct student
{
	int sno;
	char sname[20];
	char gender;
	int tfee;
	int fpaid;
};
main()
{
	FILE *fp;
	struct student s;
	char c='y';
	clrscr();
	fp = fopen("student.dat","wb");
	while(c=='y')
	{
		printf("Enter student details:\n");
		scanf("%d",&s.sno);
		fflush(stdin);
		gets(s.sname);
		fflush(stdin);
		s.gender=getchar();
		scanf("%d %d",&s.tfee,&s.fpaid);
		fwrite(&s,sizeof(s),1,fp);
		printf("Add student details...Press y or n: ");
```

```
            fflush(stdin);
            scanf("%c",&c);
      }
      fclose(fp);
      printf("\nStudent Details\n");
      fp=fopen("student.dat","rb");
      printf("SNO \t Name \t Gender \t TotalFees \t FeesPaid \t
      BalanceFee\n");
      while((fread(&s,sizeof(s),1,fp))!=0)
      {
      printf("%d \t %s \t %4c \t\t %6d \t %5d \t\t %5d\n",
            s.sno,s.sname,s.gender,s.tfee,s.fpaid,(s.tfee-s.fpaid));
      }
      fclose(fp);
      getch();
}
```

Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (fread() and fwrite()) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

RANDOM ACCESS TO FILES

So far we accessed the file sequentially, accessing from first byte to last byte. But it is also possible to access the file from a particular location.

When we are interested in accessing only a particular part of a file and not in reading other parts, it can be done with the concept of Random access. Random access allows you to move to a particular position in the file and perform input and output from that position.

When you access the file from beginning to end it is called **Sequential Access**. When you read/write file from a specific position, it is called **Random Access**. Random Access means moving the pointer of the file to the required location and read/write content at the location.

It can be achieved with the help of functions fseek(), ftell() and rewind() available in the I/O library.

ftell(): ftell takes file pointer and returns a number type long that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:
variable = ftell(file_pointer);

E.g.: n = ftell(fp);

n would give the relative effect (in bytes) of current position. This means that n bytes have already been read or written.

fseek(): It is used to move the file pointer position to a desired location within the file. It takes the following form: **fseek(file_pointer, offset, position);**

file_pointer is a pointer to the file concerned, in which the pointer is to be moved.
The **offset** is a number or variable of type long. It is the number of bytes by which pointer should move from the position. Positive value moves forward and negative value moves backward.
The **position** indicates from where the movement should take place. It can take one of the following:

0 – Beginning of file
1 – Current position
2 – End of the file

Instead of numbers the symbolic constants – **SEEK_SET, SEET_CUR, SEEK_END**, which are declared in stdio.h can also be used.

When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond file boundary an error occurs and fseek returns -1.

E.g.:

- fseek(fp, 100, 0) – Moves pointer fp forward to 100th byte from the beginning of the file. The first byte is at index 0. fseek() skips first 100 bytes (0-99) and places pointer at byte with index 100.
- fseek(fp, -25L, 1) - Moves pointer fp backward by 25 byte from current position of the file.
- fseek(fp, -10L, 2) - Moves pointer fp backward by 100 bytes from end of the file.

/*Following snippet reads fifth record from student.dat*/
```
struct student s;
FILE *fp;
/*open file in read binary mode*/
fp = fopen("student.dat","rb");
/*skip 4 record and move to 5th record*/
fseek(fp, sizeof(struct student)*4,0);
/*read a record*/
fread(&s,sizeof(struct student),1,fp);
```

EXAMPLES OF FSEEK & FTELL

FI.TXT contains the following content

H	e	l	l	o		h	i		h	o	w		a	r	e		y	o	u
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	d	e	a	r		s	t	u	d	e	n	t	s	EOF					
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34					

Example: Program to print every 5th character from beginning in a given file.

```
#include<stdio.h>
main()
{
	int n=0;
	char c;
	FILE *fp;
	clrscr();
	fp = fopen("f1.txt","r");
	while((c=getc(fp))!=EOF)
	{
		printf("%c",c);
		n=n+5;
		fseek(fp,n,0);
	}
	getch();
}
Output: h oe  e
```

Example: Program to print every 5th character from current position in a given file.

```
#include<stdio.h>
main()
{
	int n=0;
	char c;
	FILE *fp;
	fp = fopen("f1.txt","r");
	while((c=getc(fp))!=EOF)
	{
		printf("%c",c);
		fseek(fp,5,1);
	}
	getch();
}
Output: hh ore
```

Example: Program to print contents of a given file in reverse

```
#include<stdio.h>
main()
{
	int n=1,i;
	char c;
	FILE *fp;
	clrscr();
	fp = fopen("f1.txt","r");
	while((c=getc(fp))!=EOF);
```

```
     i=ftell(fp);
     fseek(fp,-n,2);
     while(i>=0)
     {
          c=getc(fp);
          printf("%c",c);
          n++;
          i--;
          fseek(fp,-n,2);
     }
     getch();
}
Output: stneduts raed uoy era woh ih olleh
```

Example: Program to print contents of file from end

```
#include<stdio.h>
main()
{
     int n=1,i;
     char c;
     FILE *fp;
     clrscr();
     fp = fopen("f1.txt","r");
     while((c=getc(fp))!=EOF);
     i=ftell(fp);
     n=i;
     fseek(fp,-(n),2);
     while(n!=0)
     {
          c=getc(fp);
          printf("%c",c);
          n--;
          fseek(fp,-n,2);
     }
     getch();
}
Output: hello hi how are you dear students
```

Example: Program to print every 5th character from end

```
#include<stdio.h>
main()
{
     int n,i;
     char c;
     FILE *fp;
     clrscr();
     fp = fopen("f1.txt","r");
     while((c=getc(fp))!=EOF);
     i=ftell(fp);
     n=1;
     fseek(fp,-n,2);
     while(n<i)
     {
          c=getc(fp);
          printf("%c",c);
          n=n+5;
          fseek(fp,-n,2);
     }
     getch();
}
Output: suaoa l
```

rewind(): It takes a file pointer and resets the position to the start of the file.

Syntax: rewind(file_pointer);

For example, the statement

rewind(fp);
n=ftell(fp);

would assign 0 to n because the file position has been set to the start of the file by rewind. The first byte in the file is numbered as 0, second as 1 and so on.

This function helps us in reading a file more than once, without having to close and open the file. Whenever a file is opened for reading or writing a rewind is alone implicitly. This function can be used for the modes of operation with + as postfix to the modes (w, a, r, wb, rb, and ab).

Example: Program to illustrate rewind() function.

```
#include<stdio.h>
main()
{
     FILE *f1;
     char c;
     clrscr();
     printf("Write data to file: \n”);
     printf("Specify @ to end the file reading \n\n");
     f1 = fopen("sample.txt","w+");
     while((c=getchar())!='@')
          putc(c,f1);
     rewind(f1);
     printf("\nRead data from file: \n");
     while((c=getc(f1))!=EOF)
          printf("%c",c);
```

```
    fclose(f1);
    getch();
}
```

Enumeration

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword enum is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, *type_name* is the name of enumerated data type or tag. And *value1*, *value2*,....,*valueN* are values of type *type_name*.

By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements
enum suit{
   club=0;
   diamonds=10;
   hearts=20;
   spades=3;
};
```

Declaration of enumerated variable

Above code defines the type of the data but, no any variable is created. Variable of type enum can be created as:

```
enum boolean{
   false;
   true;
};
enum boolean check;
```

Here, a variable check is declared which is of type **enum boolean**.

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday,
saturday};
int main(){
    enum week today;
    today=wednesday;
    printf("%d day",today+1);
```

```
     return 0;
    }
```
Output

```
4 day
```

typedef

typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

typedef *existing_name alias_name*

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

ulong i, j ;

Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
      typedef struct
      {
        type member1;
        type member2;
        type member3;
      } type_name ;
```

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

type_name t1, t2 ;

Example of structure definition using typedef

```
#include< stdio.h>
#include< conio.h>
#include< string.h>
typedef struct employee
{
 char  name[50];
 int   salary;
} emp ;
void main( )
{
 emp e1;
 printf("\nEnter Employee record\n");
 printf("\nEmployee name\t");
 scanf("%s",e1.name);
 printf("\nEnter Employee salary \t");
 scanf("%d",&e1.salary);
 printf("\nstudent name is %s",e1.name);
 printf("\nroll is %d",e1.salary);
 getch();
}
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers `*` binds to the right and not the left.

```
int* x, y ;
```

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

typedef int* **IntPtr** ;
IntPtr x, y, z;

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

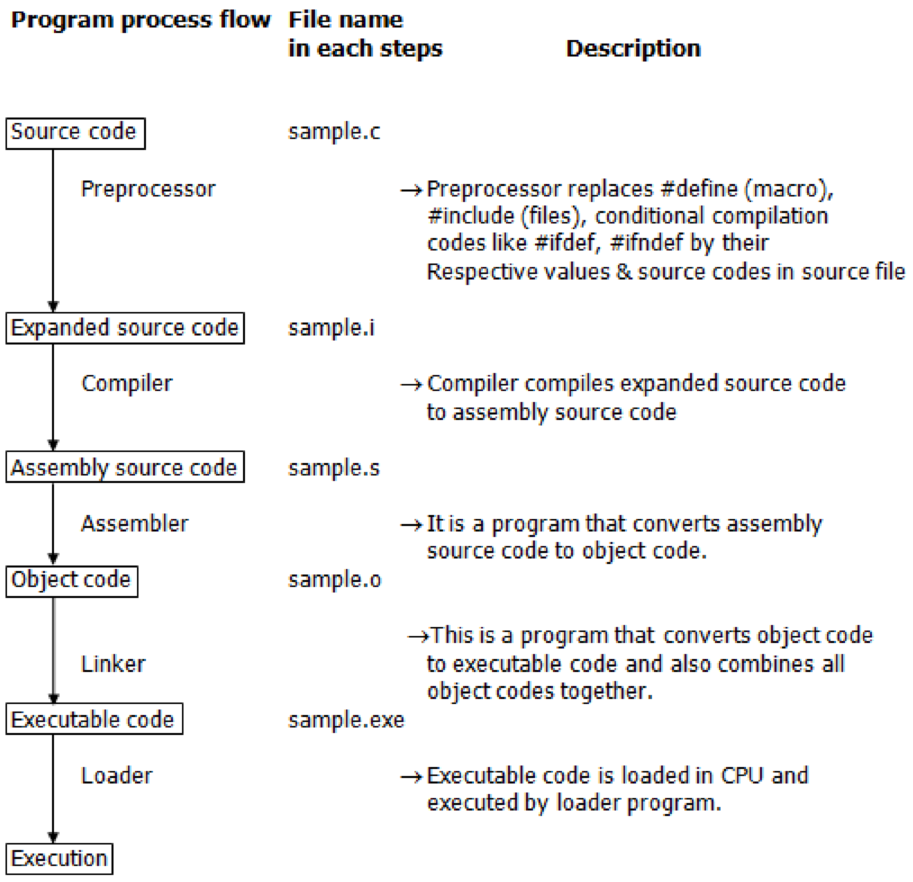
C Preprocessor directives:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.
- Below is the list of preprocessor directives that C language offers.

S.no	Preprocessor	Syntax	Description
1	Macro	#define	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	#include <file_name>	The source code of the file “file_name” is included in the main program at the specified place
3	Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Set of commands are included or excluded in source program before compilation with respect to the condition

4	Other directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.


Program process flow
File name in each steps
Description
Source code
sample.c
Preprocessor
→Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file
Expanded source code
sample.i
Compiler
→Compiler compiles expanded source code to assembly source code
Assembly source code
sample.s
Assembler
→It is a program that converts assembly source code to object code.
Object code
sample.o
Linker
→This is a program that converts object code to executable code and also combines all object codes together.
Executable code
sample.exe
Loader
→Executable code is loaded in CPU and executed by loader program.
Execution

Example program for #define, #include preprocessors in C:

- #define - This macro defines constant value and can be any of the basic data types.
- #include <file_name> - The source code of the file “file_name” is included in the main C program where “#include <file_name>” is mentioned.

```
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
void main()
{
   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
   printf("value of letter : %c \n", letter );
   printf("value of letter_sequence : %s \n", letter_sequence);
   printf("value of backslash_char  : %c \n", backslash_char);

}
```

Output:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

Example program for conditional compilation directives:

a) Example program for #ifdef, #else and #endif in C:

- "#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.
- Otherwise, "else" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100
int main()
{
   #ifdef RAJU
   printf("RAJU is defined. So, this line will be added in " \
          "this C file\n");
   #else
   printf("RAJU is not defined\n");
   #endif
   return 0;
}
```

Output:

```
RAJU is defined. So, this line will be added in this C file
```

b) Example program for #ifndef and #endif in C:

- #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

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

```
#define RAJU 100
int main()
{
   #ifndef SELVA
   {
      printf("SELVA is not defined. So, now we are going to " \
             "define here\n");
      #define SELVA 300
   }
   #else
   printf("SELVA is already defined in the program”);

   #endif
   return 0;
}
```

Output:

```
SELVA is not defined. So, now we are going to define here
```

c) Example program for #if, #else and #endif in C:

- “If” clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100
int main()
{
   #if (a==100)
   printf("This line will be added in this C file since " \
          "a \= 100\n");
   #else
   printf("This line will be added in this C file since " \
          "a is not equal to 100\n");
   #endif
   return 0;
}
```

Output:

```
This line will be added in this C file since a = 100
```

Example program for undef in C:

This directive undefines existing macro in the program.

```
#include <stdio.h>
#define height 100
void main()
```

```
{
   printf("First defined value for height    : %d\n",height);
   #undef height          // undefining variable
   #define height 600     // redefining the same for new value
   printf("value of height after undef \& redefine:%d",height);
}
```

Output:

```
First defined value for height : 100
value of height after undef & redefine : 600
```