Computer

Computer can be defined on an electronic device that accept data, process then at a high speed according to a set of instructions provide to it and produces the desired output. So computer is a programmable machine.

Generation of computer

Generation means variation between different hardware technologies

1st generation: (1946-1956)

First generation computers are made with the use of vacuum tubes. These computers used machine language for programming.

Disadvantages:

1. Occupied lot of space
2. Consumed lot of power
3. Produced lot of heat
4. Costly system

Example: ENIAC(Electronic numeric integrator and computer)
UNIVAC(Universal Accounting company)

2nd generation: (1957-1963)

The computer in which vacuum tubes were replaced by transistors are called second generation of computers these computers used assembly language for programming.

Advantages:

1. Less expensive
2. Consumed less power
3. Produced little heat
4. Less cost and work at higher speed

Example: IBM 7090, IBM 7094

3rd generation: (1964-1981)

The computers using the integrated circuits came to be known on the third generation of computers. These used high level language. Example: IBM 370, Cyber 175

4th generation: (1982-1989)

The computers which were built with microprocessor are identified on the fourth generation computers these computers use VLSI chips for both cpu and memory.
Example: CRAY-2, IBM 3090

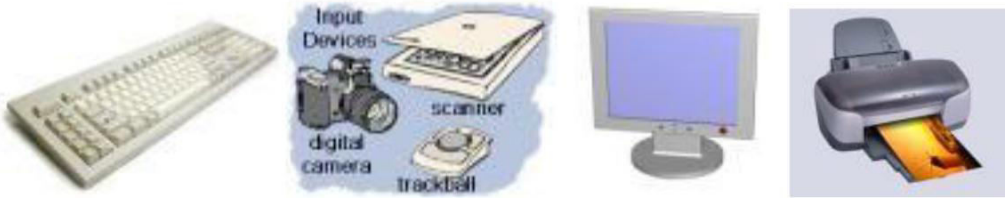
5th generation: (1990- till now)

Fifth generation computers are under development stage these computers use ULSI chips. ULSI chip contains thousands of components into a single IC. Aim of these computers is develop the artificial intelligence.

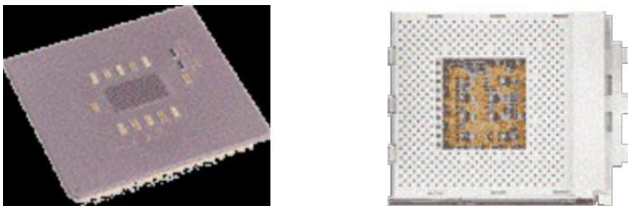
The Computer Hardware

A computer is made up of many parts:

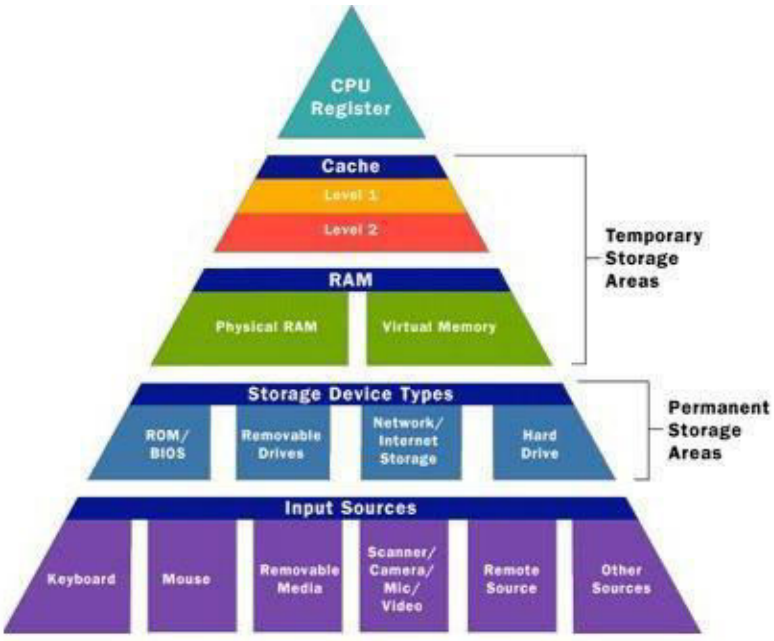
1. **Input/Output (I/O) devices** – These allow you to send information to the computer or get information from the computer.



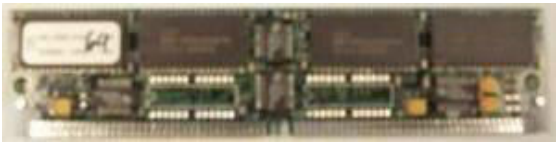
2. **Central Processing Unit** – CPU or Processor for short. The brain of a computer. Approximately 1.5 in X 1.5 in. Does all the computation/work for the computer.



3. **Memory** – Although memory is technically any form of electronic storage, it is used most often to identify fast, temporary forms of storage. Accessing the hard drive for information takes time. When the information is kept in memory, the CPU can access it much more quickly.



a. **Random Access Memory** – RAM. Where information is stored temporarily when a program is run. Information is automatically pulled into memory, we cannot control this. RAM is cleared automatically when the computer is shutdown or rebooted. RAM is volatile (non-permanent).



b. **Read Only Memory** – ROM. More permanent than RAM. Data stored in these chips is nonvolatile -- it is not lost when power is removed. Data stored in these chips is either unchangeable or requires a special operation to change. The BIOS is stored in the CMOS, read-only memory.

c. **Hard Drive** – Where you store information permanently most frequently. This is also nonvolatile.

4. **Motherboard** – A circuit board that allows the CPU to interact with other parts of the computer.

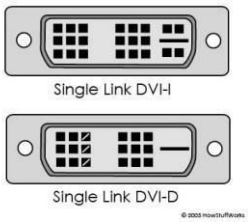
5. **Ports** – Means of connecting peripheral devices to your computer.

a. **Serial Port** – Often used to connect a older mice, older external modems, older digital cameras, etc to the computer. The serial port has been replaced by USB in most cases. 9-pin connector. Small and short, often gray in color. Transmits data at 19 Kb/s.

b. **Monitor Ports –** Used to connect a monitor to the computer.

PCs usually use a VGA (Video Graphics Array) analog connector (also known as a D-Sub connector) that has 15 pins in three rows. Typically blue in color.

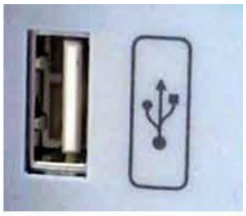
Because a VGA (analog) connector does not support the use of digital monitors, the Digital Video Interface (DVI) standard was developed.



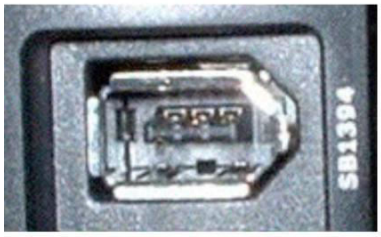
LCD monitors work in a digital mode and support the DVI format. At one time, a digital signal offered better image quality compared to analog technology. However, analog signal processing technology has improved over the years and the difference in quality is now minimal.

c. **Parallel Port** – Most often used to connect a printer to the computer. 25-pin connector. Long and skinny, often pink in color. Transmits data at 50-100 Kb/s.

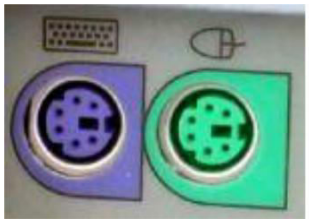
d. **USB Port** – Universal Serial Bus. Now used to connect almost all peripheral devices to the computer. USB 1.1 transmits data at 1.5 Mb/s at low speed, 12 Mb/s at full speed. USB 2.0 transmits data at 480 Mb/s.




e. **Firewire/ IEEE 1394 Port** – Often found on Apple Computers. Often used with digital camcorders. Firewire transmits data at 400 Mb/s. Firewire 1394B (the new firewire) transmits data at 3.2 Gb/s.



f. **PS/2 Port** - sometimes called a mouse port, was developed by IBM. It is used to connect a computer mouse or keyboard. Most computers come with two PS/2 ports.




g. **Ethernet Port** – This port is used for networking and fast internet connections. Data moves through them at speeds of either 10 megabits or 100 megabits or 1 gigabit (1,000 megabits) depending on what speed the network card in the computer supports. Little monitor lights on these devices flicker when in use.

6. **Power Supply** – Gives your computer power by converting alternating current (AC) supplied by the wall connection to direct current (DC).

7. **Expansion Cards –** Used to add/improve functionality to the computer.
   a. **Sound Card –** Used to input and output sound under program control. Sound cards provide better sound quality than the built in sound control provided with most computers.

   b. **Graphics Card –** Used to convert the logical representation of an image to a signal that can be used as input for a monitor.

   c. **Network Card** – Used to provide a computer connection over a network.Transmitdataat10/100/1000Mb/s.

8. **CD ROM –** A device used to read CD-ROMs. If capable of writing to the CD-ROM, then these are usually referred to as a „burner" or CD-RW.

9. **DVD ROM –** A device that is used to read DVDs/CDs. If capable of writing to the DVD, then it is often referred to as a DVD-burner or a DVD-RW.

10. **Floppy Drive –** A device that is used to read/write to floppy diskettes.

11. **Fan** – Keeps your computer cool. If the inside of your computer becomes too hot, then the computer can overheat and damage parts.

12. **Heatsink –** Used to disperse the heat that is produced inside the computer by the CPU and other parts by increasing surface area.

13. **The little parts** – Capacitors – store energy, Resistors – allows a current through, Transistors – a valve which allows currents to be turned on or off.

14. **Case** – (Tower if standing upright.) What your motherboard, CPU, etc is contained in.

The three main components of a computer:

1. **CPU** – Central Processing Unit, coordinates all actions that occur in the system, executes program instructions.
2. **I/O Devices** – Input/Output devices, which allow you to obtain or display data.
3. **Memory** – Used to store information.

Bits and Bytes:

Bits:

The term bit is an acronym of **B**inary Dig**it**. By definition, it is "a single digit in binary number scheme", meaning it can take on one of two values: 0 and 1 (a binary condition). It is a mutually exclusive state: Something either isn"t („0"), or it is („1"). It is also the basic unit of information storage.

State 0: OFF State 1: ON

Byte:

A pattern eight bits makes up a BYTE. A BYTE represents characters (letters, numbers & symbols).

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3
0	0	0	0	0	1	0	0	4
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	1	1	7
0	0	0	0	1	0	0	0	8
0	0	0	0	1	0	0	1	9

- 8 bits are 1 **byte**
- 16 bits are 2 **bytes** = 1 **halfword**
- 32 bits are 4 **bytes** = 1 **word**
- 1024 bytes are 1 **kilobyte**
- 1024 kilobytes are 1**megabyte**
- 1024 mega bytes are 1**gigabyte**
- 1024 gigabytes are 1**terabyte**
- 1024 terabytes are 1 **petabyte**
- 1024 petabytes are 1 **zettabyte**.

Programming Languages:

To write a computer program, a standard programming language is used. A programming language is composed of a set of instructions in a language understandable to the programmer and recognizable by a computer. Programming languages can be classified as high-level and low level.

Low-level programming languages were the first category of programming languages to evolve. Gradually, high-level languages were developed and put to use.

a) Low-level Languages:

A low-level computer programming language is one that is closer to the native language of the computer, which is 1‟s and 0‟s.

Machine Language:

Only computer can directly understand only its own machine language. Machine language is the natural language of a particular computer. It is defined by the hardware design of that computer. This is considered to be the *First Generation Language* (1GL).

Advantages:

The CPU directly understands machine instructions and hence no translation is required. Therefore, the computer directly starts execute the machine language instructions, and it takes less execution time.

Disadvantages:

- Difficult to use
- Machine dependent
- Error prone
- Difficult to debug and modify

Assembly Language:

When symbols as letters, digits, or special characters are employed for the operation, operand and other parts of instruction code. The representation is called an assembly language instruction. Such representations are known as mnemonic codes; they are used instead of binary codes. A program written with mnemonic codes forms an assembly language program. This is considered to be a *Second Generation Language* (2GL).

Advantages:

- Changes could be made easier and faster.
- Easier to read and write.
- Error checking is provided.

Disadvantages:

- Machine dependent.
- Programming is difficult and time consuming.
- The programmer should know all about the logical structure of the computer.

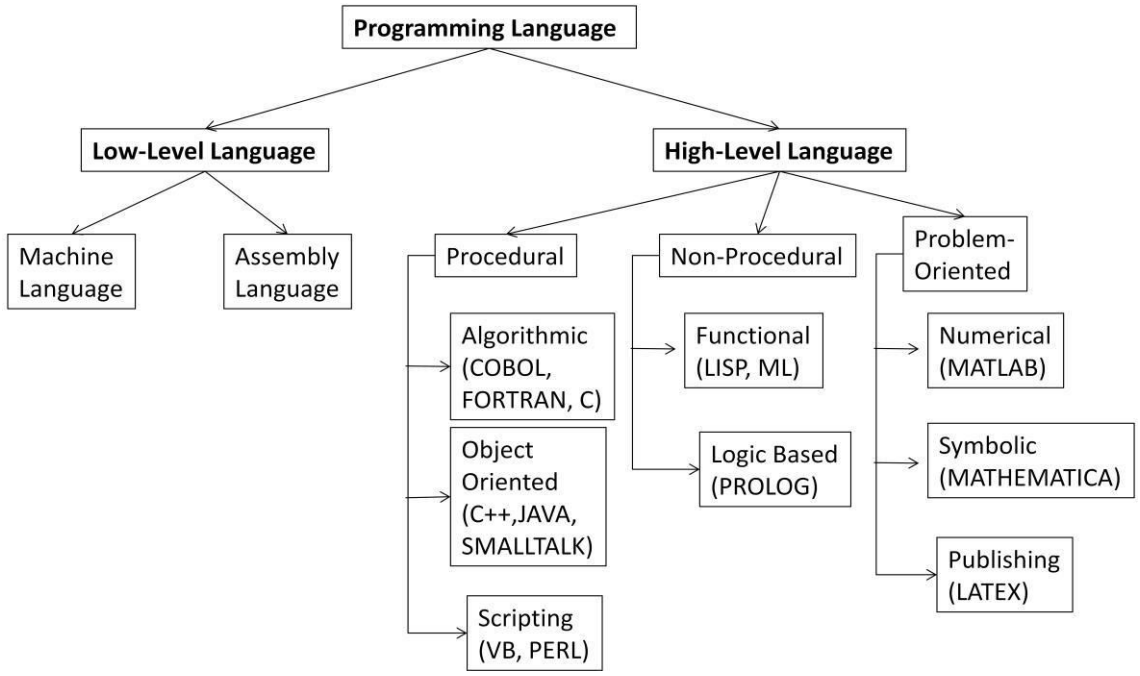


Fig: Programming Languages Classification

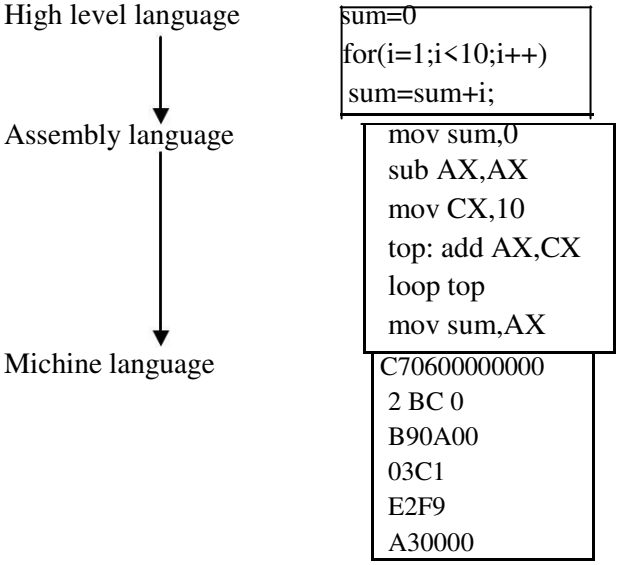
b) High-Level Languages:

Since a computer prefers machine language while a person wants to use a natural language special "high level computer languages have been developed to bridge this communication gap. A high level language like C, uses English words plus symbols and numbers. Very first two high level languages are FORTRAN and COBOL. This is considered to be a *Third Generation Language* (3GL).

Advantages:

- **Readability** Programs written in these languages are more readable than those written in assembly and machine languages.
- **Portability** High-level programming languages can be run on different machines with little or no change.
- **Easy debugging** error can be easily detected and removed.
- **Ease in the development of Software** Since commands of these programming languages are closer to the English language, software can be developed with ease.

Translation of high level language to machine language.



i) Procedural Languages:

Algorithmic Languages:

These are high-level languages designed for forming convenient expression of procedures, used in the solution of wide class of problems. In this language, the programmer must specify the steps the computer has to follow while executing a program. Some of languages that fall in the category are C, COBOL, and FORTRAN.

Object-oriented language:

The basic philosophy of object-oriented programming is to deal with objects rather than functions or subroutines as in strictly algorithmic languages. Objects are self-contained modules that contain data as well as the functions needed to manipulate the data within the same module. The difference affects the way a programmer goes about writing a program as well as how information is represented and activated in the computer. The most important object-oriented programming features are

- abstraction
- encapsulation and data hiding
- polymorphism
- inheritance
- reusable code

C++, JAVA, SMALLTALK, etc. are examples of object-oriented languages.

Scripting Languages: These languages assume that a collection of useful programs, each performing a task, already exists. It has facilities to combine these components to performing a complex task. A scripting language may thus be thought of as a glue language, which sticks a variety of components together. One of the earliest scripting languages is the UNIX shell. Now there are several scripting languages such as VB script and Perl.

ii) Problem-oriented languages

Problem-oriented Languages were designed to solve specific problems e.g. MATLAB is used to design circuit diagrams, and allowed the programmer to concentrate more on the problem rather than spending time learning the complex syntax of the language.

iii) Non-procedural Languages

Functional languages: These functional languages solve a problem by applying a set of functions to the initial variables in specific ways to get the answer. LISP, ML, etc. are examples of functional languages.

Logic Based Programming Language: A logic program is expressed as a set of atomic sentences, known as fact, and horn clauses, such as if-the rules. A query is then posed. The execution of the program now begins and the system tries to find out if the answer to the query is true or false. Such languages include PROLOG.

Software:

A set of computer programs are called software. Computer program is a series of instructions telling the computer what to do.

Types of software:

1. System software
2. Application software

1. System software:

- System software exists in the functioning of a computer system and includes the operating system, assembler, interpreter, compiler, linker and loader.
- Operating system is the interface between user applications and system hardware.
- Assembler is a translator that converse assembly language code into machine language.
- Interpreter converts source language program into executable code at once.
- Linker performs the important task of linking together several objects modules.
- The task of loading the linked object modules is performed by the loader.

System software includes three types of programs:

- **Operating System**: The combination of a particular hardware configuration and system software package is known as a computer system platform. System platforms are commonly termed as operating system (OS). Some common operating systems are DOS, UNIX, Mac, and Windows platform.
- **Language Translators**: These are interpreters and compilers for programs such as Pascal, BASIC, COBOL, C, and C++.
- **Common Utility Programs**: Communication tools, disk formatter, etc.

Examples: Language Translator, Operating System, Special Purpose Program, Utilities

2. Application software:

Application software is written to enable the computer to solve a specific data processing task. There are two categories of application software: pre-written software packages and user application programs.

A number of powerful application software packages that do not require significant programming knowledge have been developed. These are easy to learn and use compared to programming languages.

Examples:

- Database Management Software
- Spreadsheet Software
- Word processing, Desktop Publishing (DTP), and Presentation Software
- Multimedia Software
- Data Communication Software
- Statistical and operational research Software

Algorithms:

Computer Scientist „Niklaus Wirth" stated that

Programs = Algorithms + Data

An algorithm is a part of the plan for the computer program; an algorithm is „an effective procedure for solving problem in a finite number of steps".

A sequential solution of any program that written in human language, called algorithm.

Algorithm is first step of the solution process, after the analysis of problem, programmer writes the algorithm of that problem.

Good Qualities of algorithm:

- Inputs and outputs should be defined precisely.
- Each step in algorithm should be clear and unambiguous.
- Algorithm should be most effective among many different ways to solve a problem.
- Algorithm should be complete or finiteness.
- Algorithm should be correctness.

Steps for algorithm:

- Identify the processes
- Identify the major decisions
- Identify the loops
- Identify the variables

Example Algorithm for Making a Tea:

1. Put the kettle in the stove.
2. Fill the kettle with water, sugar and tea powder.
3. Boil the water in the kettle.
4. Add milk to the decoction.
5. Filter the tea into cup.
6. Drink the tea.

1. Example Algorithm for Addition of two numbers:

   Step 1: START
   Step 2: Read a, b
   Step 3: Sum = a + b
   Step 4: PRINT Sum
   Step 5: STOP

2. Example Algorithm for Biggest of two numbers:

   Step 1: START
   Step 2: Read a, b
   Step 3: IF a>b Then PRINT “a is Big” ELSE
   IF a<b Then PRINT “b is Big”
   ELSE PRINT “both are equal”
   Step 4: STOP

3. Example Algorithm for find Leap Year:

   Step 1: START
   Step 2: Read year
   Step 3: IF year%4 == 0 Then PRINT “Leap year”
   ELSE PRINT “NOT Leap Year”
   Step 4: STOP

4. Example Algorithm for Increment of Employee salary:

```
Step 1: START
Step 2: Read salary
Step 3: IF salary is greater than 5000
            THEN step 4
            ELSE IF salary is equal to 5000
            Then step5
            ELSE step 6
Step 4: Salary = Salary + (0.05*Salary)
Step 5: Salary = Salary + 200
Step 6: Salary = Salary
Step 7: Print Salary
Step 8: STOP
```

5. Example Algorithm for Factorial of Number:

```
Step 1: START                                   Step 1: START
Step 2: Read n                                  Step 2: Read n
Step 3: [Initialize] i = 1, fact = 1            Step 3: [Initialize] fact = 1
Step 4: WHILE i<=n                              Step 4: WHILE n>=1
          BEGIN                                           BEGIN
            fact = fact * i          (or)                 fact = fact * n
            i = i + 1                                     n = n - 1
          END                                             END
Step 5: PRINT fact                              Step 5: PRINT fact
Step 6: STOP                                    Step 6: STOP
```

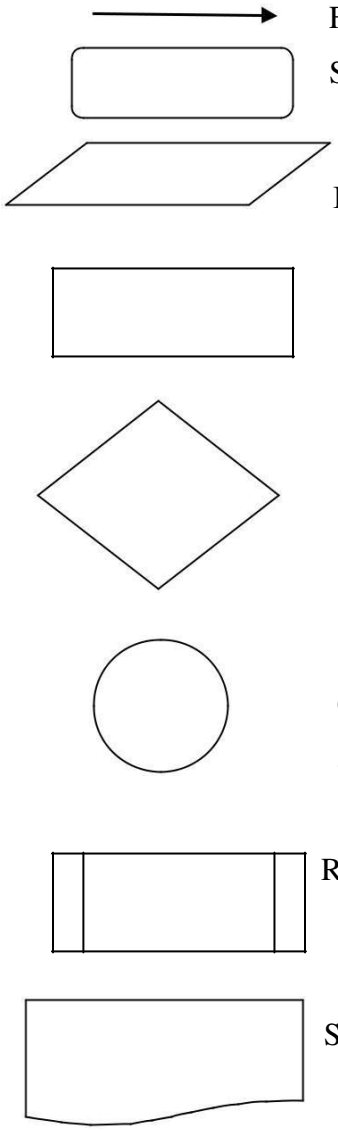
6. Example Algorithm for Swapping of Two Numbers:

```
Step 1: START                                   Step 1: START
Step 2: Read a, b                               Step 2: Read a, b
Step 3: c = a                                   Step 3: a = a + b
Step 4: a = b                (or)               Step 4: b = a - b
Step 5: b = c                                   Step 5: a = a - b
Step 6: PRINT a, b                              Step 6: PRINT a, b
Step 7: STOP                                    Step 7: STOP
```

Flowchart:

A flowchart is a pictorial representation of on algorithm. It shows the logic of the algorithm and the flow of control. The flowchart uses symbols to represent specific actions and arrows to indicate the flow of control.

Flowchart symbols are



Flow of control

Start or stop

Input or output

Process / action

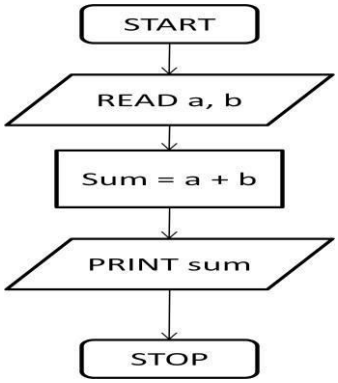
Condition, Decision, loop

Connector (Connect one part of the flow chart to another)

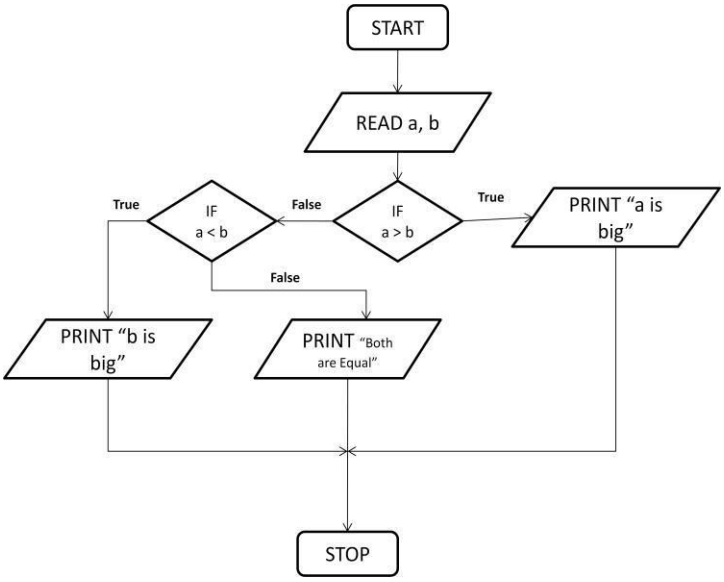
Refers to separate flowchart

Secondary storage

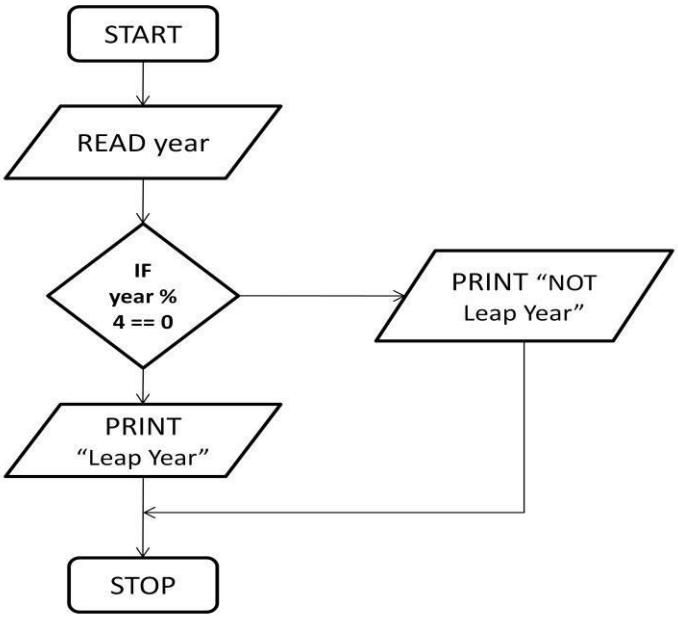
1. Example Flowchart for Addition of two numbers:



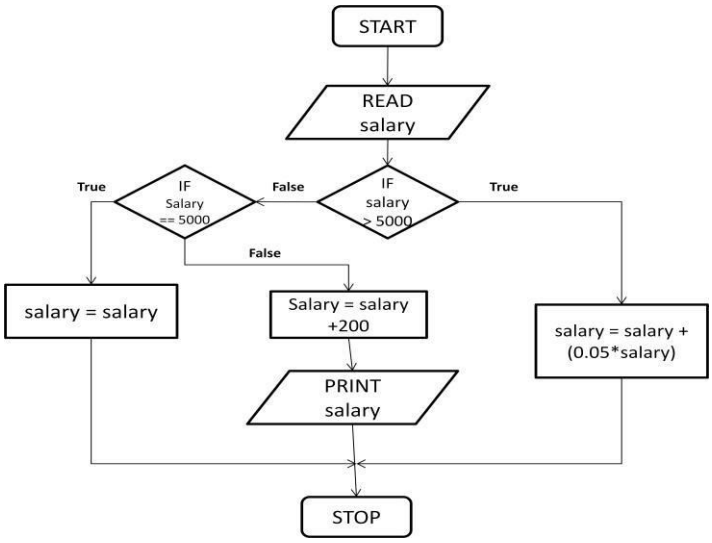
2. Example Flowchart for Biggest of two numbers:



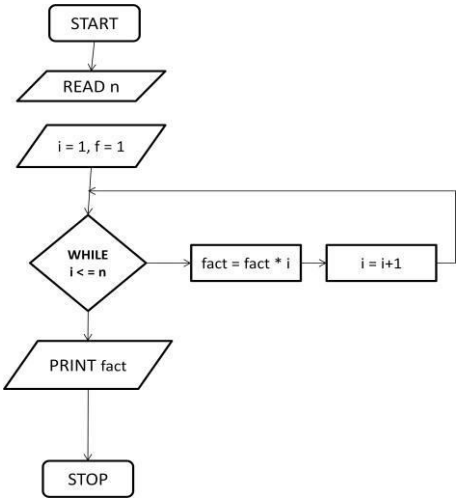
3. Example Flowchart for find Leap Year:



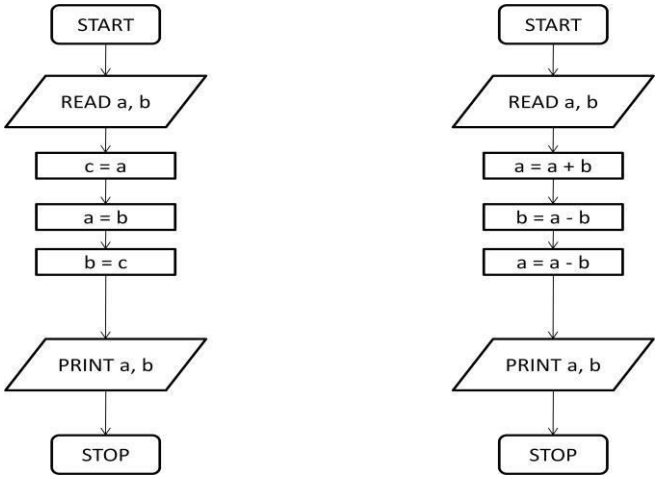
4. Example Flowchart for Increment of Employee salary:



5. Example Flowchart for Factorial of Number:



6. Example Flowchart for Swapping of Two Numbers:



The Software Development Process:

Software development Process is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

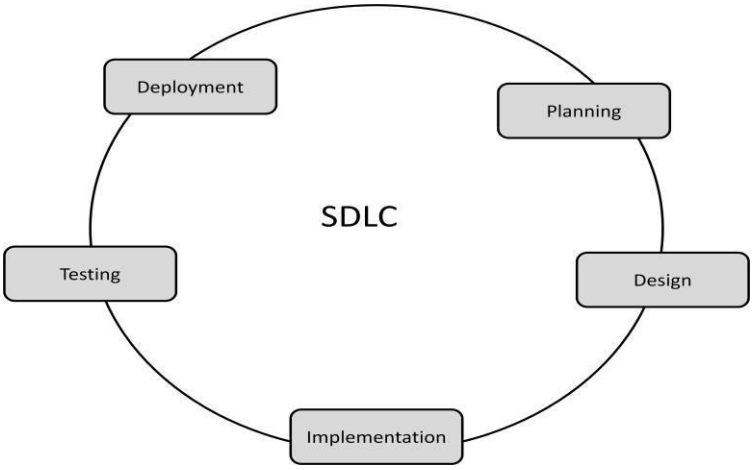


Fig: Software development Life Cycle

A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis:

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Stage 2: Designing the product architecture:

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification. This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

Stage 3: Implementing the Product:

In this stage of SDLC the actual development starts and the product is implementing. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle. Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 4: Testing the Product:

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 5: Deployment in the Market and Maintenance:

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

UNIT-2

Program development steps:

Program development contains following steps:

i). Creating the program
ii). Compiling the program
iii). Linking the program
iv). Executing the program

i) Creating the program:

In this step, the program can be written into a file through text editor. The file is saved on the disk with and extension. Corrections to the program at later stages are done to these editors. Once the program has written, it requires to be translated into machine language.

ii) Compiling the program:

This step is carried out by a program or compiler. Compiler translates the source code into object code.

Compilation of these cannot proceed successfully until and useless the source code is error free. Compiler generates messages if it encounters some errors in the source code. The error free source code translates in the object code and stored in a separate code with an extension .obj.

iii) Linking the program:

The linker links all the files and functions with the object code under execution.

Ex: printf-the linker links the user programs object with the object of the printf function. Now object code is ready for next phase.

iv) Execution the program:

The execution object code is loaded into the memory and the program execution begins. It encounters error to till the execution phase, Even though compilation phase is successful. These errors may be logical errors.

C language:

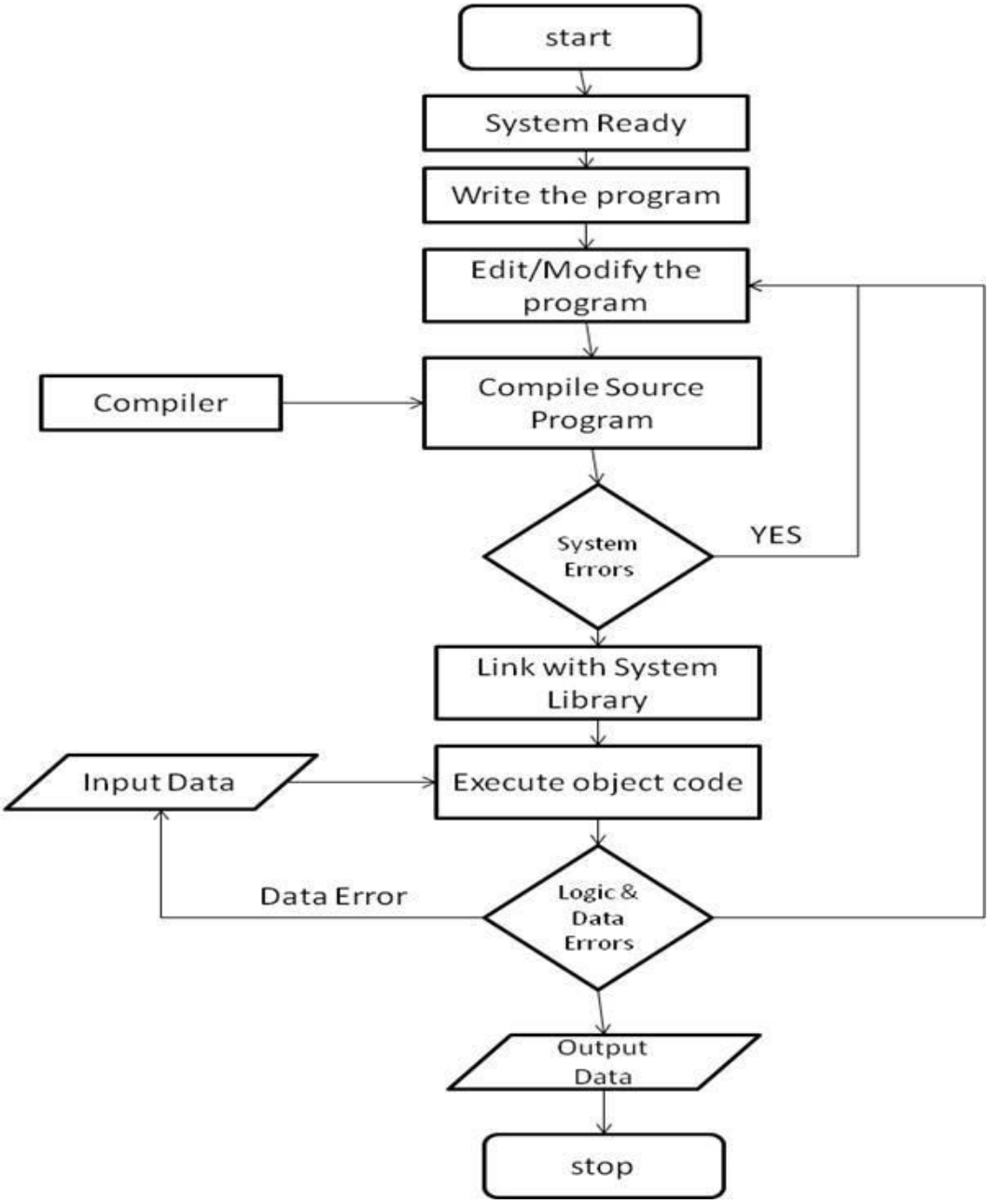
- C is case-sensitive language.
- C is a structured programming language.
- C is mother language.

Overview of "C":

- _C' seems a strange name for programming language. For this strange sounding language is one of the most popular words today.
- _C' was an off string of the ―Basic Command Programming Language (BCPL)‖ called _B', developed in the 1960 at Cambridge University. _B' language was modified by Dennis ritchi and was implemented at bell laboratories in 1972. A new language was named _C'.
- Since it was developed along with be UNIX OS, it is strongly associated with UNIX. This OS, which was also developed at bell laboratory, was coded almost entirely in _C'.

Advantages of C languages:

- _C‘ is portable, means you say program write in one computer may run on another computer successfully.
- _C‘ is fast, means that the executable program obtained after compiling linking runs very fast.
- _C‘ is compact, means that the statements written in _C‘ language or generally sharp written on very powerful.
- _C‘ language is both simplicity of high level language, low level language, and middle level language.



Structure of C:

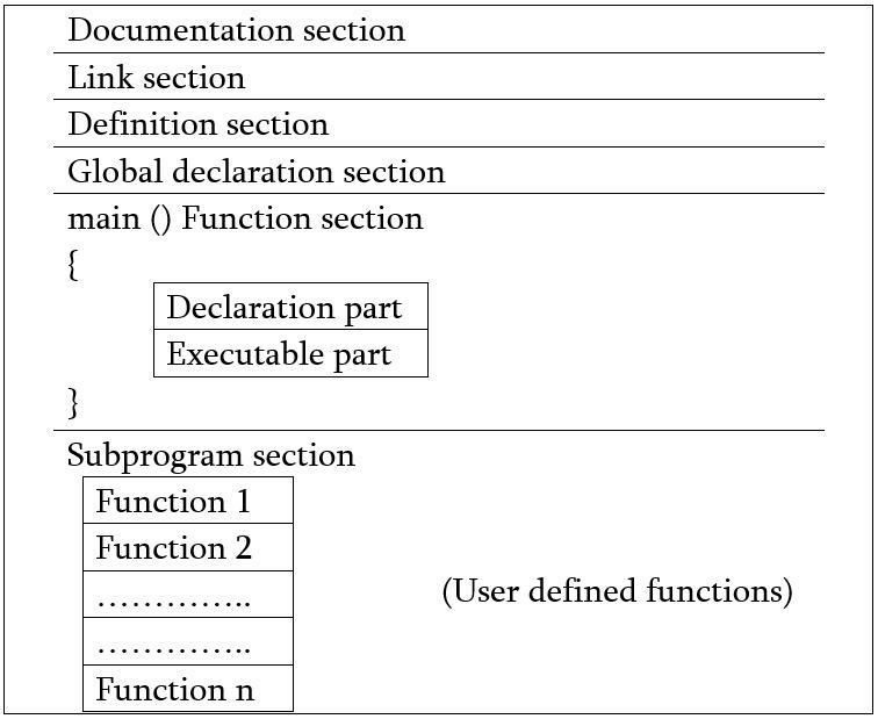


Fig: Structure of C

1. **Documentation section:** The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
2. **Link section:** The link section provides instructions to the compiler to link functions from the system library such as using the *#include directive*.
3. **Definition section:** The definition section defines all symbolic constants such using the *#define directive*.
4. **Global declaration section:** There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the *user-defined functions*.
5. **main () function section:** Every C program must have one main function section. This section contains two parts; declaration part and executable part
    1. **Declaration part:** The declaration part declares all the *variables* used in the executable part.
    2. **Executable part:** There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The *program execution* begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.
6. **Subprogram section:** If the program is a *multi-function program* then the subprogram section contains all the *user-defined functions* that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

Character set:

Character set is a set of alphabets, letters and some special characters that are valid in C language.

Alphabets:

Uppercase: A B C ..... X Y Z

Lowercase: a b c ..... x y z

Digits:

0 1 2 3 4 5 6 7 8 9

Special Characters:

,	<	>	.	_
(	)	;	$	:
%	[	]	#	?
_	&	{	}	—
^	!	*	/	|
-	/	~	+	

White space Characters, blank space, new line, horizontal tab, carriage return and form feed

Keywords:

Keywords are predefined; reserved words used in programming that have special meaning. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, **int** is a keyword that indicates **'money'** is a variable of type integer.

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

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

Along with these keywords, C supports other numerous keywords depending upon the compiler. All these keywords, their syntax and application will be discussed in their respective topics

Identifiers:

Identifiers are the names you can give to entities such as variables, functions, structures etc. identifier names must be unique. They are created to give unique name to a C entity to identify it during the execution of a program. For example:

```
int money;
double balance;
```

Here, **money** and b**alance** are identifiers.

Also remember, identifier names must be different from keywords. You cannot use **int** as an identifier because **int** is a keyword.

Rules for writing an identifier:

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscore only. For ex., a variable named _NUMBER is not the same as the variable named _number‘ and neither of them is the same sa the variable named _Number‘. All three refer to different variables.
2. The underscore should **not** be used as the first character of a variable name because several compiler-defined identifiers in the standard C library have the underscore for beginning character, and it can conflict with system names.
3. A numeric digit should **not** be used as the first character of an identifier.
4. There is no rule on the length of an identifier. However, the first 31 characters of identifiers are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.
5. The identifier should not be a keyword of C.

The main() Function:

All C language programs must have a main() function. It‘s the core of every program. It‘s required. The main() function doesn‘t really have to do anything other than be present inside your C source code. Eventually, it contains instructions that tell the computer to carry out whatever task your program is designed to do. But it‘s not officially required to do anything. When the operating system runs a program in C, it passes control of the computer over to that program. This is like the captain of a huge ocean liner handing you the wheel. Aside from any fears that may induce, the key point is that the operating system needs to know where inside your program the control needs to be passed. In the case of a C language program, it‘s the main() function that the operating system is looking for.

At a minimum, the main() function looks like this:

```
main ( )
{
}
```

Like all C language functions, first comes the function‘s name, main, then comes a set of parentheses, and finally comes a set of braces, also called curly braces.

If your C program contains only this line of code, you can run it. It won‘t do anything, but that‘s perfect because the program doesn‘t tell the computer to do anything. Even so, the operating system found the main() function and was able to pass control to that function which did nothing but immediately return control right back to the operating system. It‘s a perfect, flawless program.

The set of parentheses after a C language function name is used to contain any arguments for the function — stuff for the function to digest. For example, in thesqrt() function, the parentheses hug a value; the function then discovers the square root of that value.

The main ( ) function uses its parentheses to contain any information typed after the program name at the command prompt. This is useful for more advanced programming. Beginning programmers should keep in mind what those parentheses are there for, but you should first build up your understanding of C before you dive into that quagmire.

The braces are used for organization. They contain programming instructions that belong to the function. Those programming instructions are how the function carries out its task or does its thing.

By not specifying any contents, as was done for the main() function earlier, you have created what the C Lords call a dummy function — which is kind of appropriate, given that you‘re reading this at Dummies.com.

The printf ( ) Function:

It is a library function. It is provided by the compiler, ready for use. In addition to its variable, a function, including main ( ), may optionally have arguments those are listed in the parenthesis following the function name.

In the C Programming Language, the **printf function** writes a formatted string to the *stdout* stream.

Syntax:

The syntax for the printf function in the C Language is:

printf(―Control_String‖ ,

list_of_variables); **Ex:-** printf(―Hello‖);

When the printf ( ) function is executable its built-in instructions process this argument. The result is that the string is display on the output device, usually assumed as a display screen terminal.

The output is

Hello

- C uses a semicolon as a statement terminator; the semicolon is required as a signal to the compiler to indicate that a statement is complete.
- All program instructions, which are also called statements, have to be written in lowercase characters.
- The following statement implies the preprocessor directive:
  #include<stdio.h>
- The #include directive includes the contents of a file during compilation. In this case the file stdio.h is added in the source program before the actual compilation begins.

Escape sequence:

The \n (pronounced backslash n) in the string argument of the function printf().

printf(―Welcome\nComputer‖);

is an example of escape sequence.

It is used print the new line character. If the program is executed, the \n does not appear in the output. Each \n in the string argument of a prinf() causes the cursor to be placed at the beginning of the next line of output.

Sno	Code	meaning
1	\a	Ring terminal bell (a is for alert)
2	\?	Question mark
3	\b	Backspace
4	\r	Carriage return
5	\f	Form feed
6	\t	Horizontal tab
7	\v	Vertical tab
8	\o	ASCII null character
9	\\	Back slash
10	\‖	Double quote
11	\‘	Single quote
12	\n	New line
13	\o	Octal constant
14	\x	Hexadecimal constant

Format Modifiers:

Describes the output as well as provides a placeholder to insert the formatted string. Here are a few examples:

Format	Explanation	Example
%c	Display Character	m
%s	Display group of characters	hai
%d (%i)	Display an integer	10
%ld	Display long decimal	51200
%o	Display Octal value	12
%u	Display unsigned integer	15
%x	Display hexa value	b
%X	Display HEXA value	B
%f	Displays a floating-point number in fixed decimal format	10.500000
%.1f	Displays a floating-point number with 1 digit after the decimal	10.5

Format	Explanation	Example
%e	Display a floating-point number in exponential (scientific notation)	1.050000e+01
%g	Display a floating-point number in either fixed decimal or exponential format depending on the size of the number (will not display trailing zeros)	10.5

COMMENT:

A "comment" is a sequence of characters beginning with a forward slash/asterisk combination (**/***) that is treated as a single white-space character by the compiler and is otherwise ignored. A comment can include any combination of characters from the represent able character set, including newline characters, but excluding the "end comment" delimiter (***/**). Comments can occupy more than one line but cannot be nested.

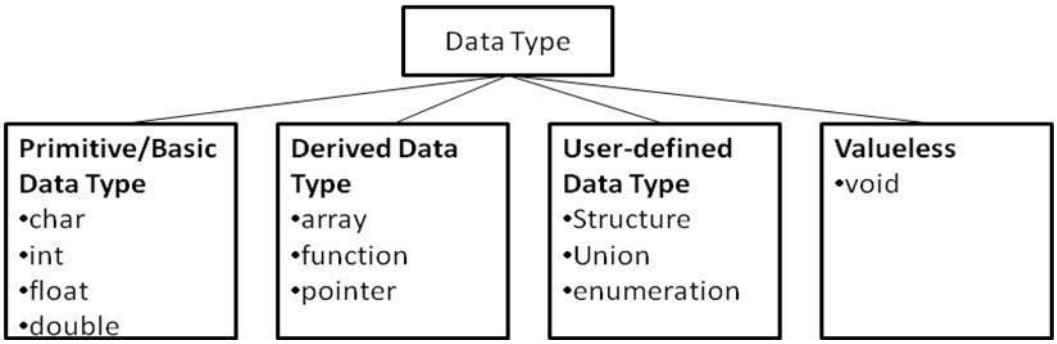
Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. The compiler ignores the characters in the comment.

Use comments to document your code. This example is a comment accepted by the compiler:

```
/* Comments can contain keywords such as
   for and while without generating errors. */
```

Data types of C:

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.



a) Primitive/Basic Data Type:

Integer data types:

C offers 3 different individual data types they are int, short int, long int. the difference between these 3 individual types are bytes required and range of values.

TYPE	SIZE (bits)	RANGE
short int	16	-2^{15} to $2^{15}-1$
int	16	-2^{15} to $2^{15}-1$
long int	32	-2^{31} to $2^{31}-1$
unsigned short int	16	0 to $2^{16}-1$
unsigned int	16	0 to $2^{16}-1$
unsigned long int	32	0 to $2^{32}-1$

Float data types:

Like integer, float are divided into 3 different individual data types they are float, double, long double. The differences between these 3 individual types are bytes required and range of values.

TYPE	SIZE (bits)	RANGE
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 3.4E+4932

Character data types:

Keyword **char** is used for declaring character type variables.

For example: char a = ‗m‘;

TYPE	SIZE (bits)	RANGE
char	8	-128 to 127
unsigned char	8	0 to 255

b) User Defined Data Type:

The user defined data types are two types. They are

1. Type definition
2. Enumerated data type

1. Type definition:

The users can define and identifies that represents an existing data types the users as type definition.

Ex: typedef int sno;

sno c1,c2;

2. Enumerated data type:

User defined can be used to declare variables that can have one of the values out of many enumerations constant. After that the user can declare variables to be of this new type.

```
#include<stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday,
friday, saturday}; main ( ){
  enum week today;
  today=wednesday;
  printf("%d day",today);
}
```

OPERATORS:

An operator is a symbol which represents a particular operation that can be performed on data.

The data itself (which can be either a variable or constant) is called operand. Expressions made by combining operators and operands.

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Unary operators
5. Bitwise operators
6. Logical/Boolean operators
7. Conditional operators
8. Special operators

1. Arithmetic operations:

The arithmetic operators in _C' language are + (addition), - (subtraction), * (multiplication), / (division), % (modulus) these operators are called arithmetic/binary operators. Each operand can be int, float, char.

Ex: x+y, x-y, x*y, x/y, x%y.

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

2. Assignment operators:

These are used to assign the result of the expression to a variable, assignment operator is _=' Syntax: V op= Exp

V- variable
op-arithmetic operator
Exp-Expression

+= add assignment
-= minus assignment
*= multiply assignment
/= divide assignment
%= Modulus assignment

3. Relational operators:

The relational operators are used to test or compare the values between two operands. The relational or equality operators produce an integer value to express the condition of the comparison. If the condition is false then the integer result is _0‘. Otherwise, the integer result is nonzero.

<	less than	= =	equal to
< =	less than or equal to	! =	not equal to
>	greater than	=	assignment
> =	greater than or equal to		

4. Unary operators:

The unary _++‘ and _- -_operators increment or decrement the value in a variable by 1. There are _pre‘ and _post‘ variants for both operators that do slightly different things as explained below.

Var ++ increment _post‘ variant
++Var increment _pre‘ variant
Var - - decrement _post‘ variant
- -Var decrement _pre‘ variant

i) x=a++; → x=a; a=a+1;
ii) x=a- -; → x=a; a=a-1;
iii) x=++a; → a=a+1; x=a;
iv) x=- -a; → a=a-1; x=a;

5. Bitwise operators:

A smallest element in the memory on which they are able to operator is a bit, C suppose several bitwise operators.

& Bitwise AND
| Bitwise OR
>> Bitwise Right Shift
<<Bitwise Left Shift
^ Bitwise Exclusive OR
~ Bitwise Negation

Ex: x=7, y=8
z=x & y; =0
Ex: x=7, y=8
z=x | y;=15
Ex: x=7, y=8
z=x ^ y; =15
Ex: x=7
z=x >> 3
Ex: x=7
z=x << 2
Ex: x=1
z= ~ x;

6. Logical/Boolean operators:

These operators are used to compare two or more operands. The logical operator is also called unary operators.

! Logical NOT

&& Logical AND

|| Logical OR

Example	Result
! (5= =5)	0
(5<6) && (6<5)	0
(5<6) || (6<5)	1

7. Conditional operators:

The conditional operator **?** and **:** are sometimes called ternary operator since it operands on three operands, and it is consider has IF-THEN-ELSE in C statement.

```
#include<stdio.h>
main ( )
{
        int a=7,b=8;
        printf ( (a<b) ? "a is big" :  "b is big"       );
}
```

8. Special operators:

Some commonly used special operators are comma operator (**,**), sizeof operator, address operator (**&**) and value of address operator (*****).

i) Comma operator:

The operator permits two different expressions, appears in a situation where only one expression put ordinary be use. The expressions are separated by comma operator. Ex: int a, b;

int c = (a=10, b=20, a+b);

ii) Sizeof operator:

The size of operator returns the number of bytes occupied in memory by operand may be the variable or constant or datatype.

Ex: sizeof (int);

iii) Address operator:

The address of the operator (**&**) returns the address of the variable.

Ex: printf (― Address is %d ‖ , &a);

iv) Value of address operator:

The value at address operator (*****) return the value stored at particular address.

Ex: x = * n;

Ex:

```
#include <stdio.h>
main() {
  int a = 21;
  int b = 10;
  int c ;

  c = a + b;
  printf("Line 1 - Value of c is %d\n", c );

  c = a - b;
  printf("Line 2 - Value of c is %d\n", c );

  c = a * b;
  printf("Line 3 - Value of c is %d\n", c );

  c = a / b;
  printf("Line 4 - Value of c is %d\n", c );

  c = a % b;
  printf("Line 5 - Value of c is %d\n", c );

  c = a++;
  printf("Line 6 - Value of c is %d\n", c );

  c = a--;
  printf("Line 7 - Value of c is %d\n", c );
}
```

When you compile and execute the above program, it produces the following result

```
− Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

Variable:

A variable is an identifier for a memory location in which data can be stored and subsequently recalled. All the variables have two important attributes.

A data type that is established when the variable is defined, e.g., integer, real, character. Once defined, the type of a C variable cannot be changed.

A value can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For e.g., an integer variable can only take integer values, e.g., 2, 100, -12, -14, 0, 4.

Declaration of variables:

Any programming language any variable used in the program must be declared before it is used. This declaration tells the compiler what the variable name and what type of data it is.

In C language a declaration of variable should be done in the declaration part of the program. A type declaration statement is usually written at the beginning of the program.

Syntax: <data_type> <var1>,<var2>,<var3>,……,<var n>;

Ex: int i, count;
float price, area;
char c;

Assigning values:

Values can be assigned to variables by using the assignment operator. An assignment statement employees that the value of the variable on the left of the equal to the value of quantity on the right.

Syntax: <data_type> variable_name = constant;

Ex: int i = 20;

Declaration of statements:

In computer programming, a **declaration** specifies properties of an identifier: it declares what a word (identifier) *means*. Declarations are most commonly used for functions, variables, constants and classes, but can also be used for other entities such as enumerations and type definitions.

Here are some examples of declarations that are not definitions, in C:

```
extern char example1;
extern int example2;
void example3(void);
```

Here are some examples of declarations that are definitions, again in C:

```
char example1; /* Outside of a function definition it will be initialized to zero. */
int example2 = 5;
void example3(void) { /* definition between braces */ }
```

Expression:

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2 +2x + 1$	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and 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.

Example of evaluation statements are

```
x = a * b – c
y = b / c * a
z = a – b / c + d;
```

The following program illustrates the effect of presence of parenthesis in expressions.

Ex:

```
main ()
{
        float a, b, c x, y, z;
        a = 9;
        b = 12;
        c = 3;
        x = a – b / 3 + c * 2 – 1;
        y = a – b / (3 + c) * (2 – 1);
        z = a – ( b / (3 + c) * 2) – 1;
        printf (—x = %fn‖,x);
        printf (—y = %fn‖,y);
        printf (—z = %fn‖,z);
}
```

Output:

```
x = 10.00
y = 7.00
z = 4.00
```

Operator Precedence:

This page lists C operators in order of precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
( ) [ ] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.,* **z++** evaluates to **z**) and **z** only incremented after all else is done.

Type Conversions:

There are two kinds of type conversion we need to talk about: automatic or **implicit** type conversion and **explicit** type conversion.

1. Implicit Type Conversion

The operators we have looked at can deal with different types. For example we can apply the addition operator + to an int as well as a double. It is important to understand how operators deal with different types that appear in the same expression. There are rules in C that govern how operators convert different types, to evaluate the results of expressions.

For example, when a floating-point number is assigned to an integer value in C, the decimal portion of the number gets truncated. On the other hand, when an integer value is assigned to a floating-point variable, the decimal is assumed as .0.

This sort of implicit or automatic conversion can produce nasty bugs that are difficult to find, especially for example when performing multiplication or division using mixed types, e.g. integer and floating-point values. Here is some example code illustrating some of these effects:

```
#include <stdio.h>
int main() {
    int a = 2;
    double b = 3.5;
    double c = a * b;
    double d = a / b;
    int e = a * b;
    int f = a / b;
    printf("a=%d, b=%.3f, c=%.3f, d=%.3f, e=%d, f=%d\n",a, b, c, d, e, f);
    return 0;
}
```

Output:

a=2, b=3.500, c=7.000, d=0.571, e=7, f=0

2. Explicit Type Conversion

Type Casting

There is a mechanism in C to perform **type casting** that is to force an expression to be converted to a particular type of our choosing. We surround the desired type in brackets and place that just before the expression to be coerced. Look at the following example code:

```
#include <stdio.h>
#include <stdio.h>
int main() {
    int a = 2;
    int b = 3;
    printf("a / b = %.3f\n", a/b);
    printf("a / b = %.3f\n", (double) a/b);
    return 0;
}
```

Mathematical library functions:

The <math.h> header file contains the mathematical operations like trigonometric and other mathematic formats.

Function	Return type	Description
acos(d)	double	Return the $\cos^{-1}$ of d
asin(d)	double	Return the $\sin^{-1}$ of d
atan(d)	double	Return the $\tan^{-1}$ of d
atan2(d1,d2)	double	Return the $\tan^{-1}$ of d1/d2
ceil(d)	double	Return a value rounded upto the next higher integer
cos(d)	double	Return the cos of d
cosh(d)	double	Return the hyperbolic cos of d
exp(d)	double	Return e of the power of d
fabs(d)	double	Return absolute value of d
floor(d)	double	Return a value rounded downto the next lower integer
fmod(d1,d2)	double	Return the remainder of d1/d2 (with same sign as d1)
labs(l)	long int	Return the absolute value of l
log(d)	double	Return natural logarithm of d
log10(d)	double	Return natural logarithm (base 10) of d
pow(d1,d2)	double	Return d1 raised to the d2 power ($d1^{d2}$)
sin(d)	double	Return the sin of d
sinh(d)	double	Return the hyperbolic sin of d
sqrt(d)	double	Return square root of d
tan(d)	double	Return the tan of d
tanh(d)	double	Return the hyperbolic tan of d

Basic Screen and keyboard I/O in C:

C provides several functions that give different levels of input and output capability. These functions are, in most cases, implemented as routines that call lower-level input/output functions.

The input and output functions in C are built around the concept of a set standard data streams being connected from each executing data streams or files are opened by the operating system and are available to every C and assembler program to use without having to open or close the files. These standard files or streams are called

stdin: Connected to the keyboard
stdout: Connected to the screen
stderr: Connected to the screen

The following two DataStream‘s are also available on MSDOS-based computers, but not an UNIX or other multi-user-based operating systems.

stdaux: Connected to the first serial communication
stdprn: Connected to the first parallel printer port

i) Non –formatted Input and Output
ii) Formatted Input and Output

i) Non-formatted Input and Output:

Non-formatted input and output can be carried out by standard input-output library functions in C. these can handle one character at a time.

For the input functions, it reads the character. For the output functions, it prints the single character on the screen.

a) Single character input:

The getchar( ) input function reads an unsigned char from the input stream stdin. To read a single character from the keyboard, the general form of the statement used to call the getchar() function is given as follows:

char_variable_name = getchar( );

where char_variable_name is the name of a variable of type char. The getchar() input function receives the character data entered, through the keyboard, and places it in the memory location allotted to the variable char_variable_name.

Ex:

```
#include<stdio.h>
main()
{
        char ch;
        int a,b,c;
        printf(―Would you like perform addition or subtraction\?\n‖);
        printf(―Type A for addition and S for subtraction ‖);
        ch = getchar( );
```

```
                if(ch==‘A‘ || ch==‘a‘)
                {
                        printf(—\nEnter a and b values: ‖);
                        scanf(—%d %d‖, &a, &b);
                        c = a+b;
                        printf(—\n Addition is %d‖,c);
                }
                else if(ch==‘A‘ || ch==‘a‘)
                {
                        printf(—\nEnter a and b values: ‖);
                        scanf(—%d %d‖, &a, &b);
                        c = a-b;
                        printf(—\n Subtraction is %d‖,c);
                }
                else
                {
                        Printf(—\nInvalid letter‖);
                }
        }
```

Output1:

Would you like perform addition or subtraction? Type A for addition and S for subtraction A Enter a and b values: 5 6 Addition is 11

Output2:

Would you like perform addition or subtraction? Type A for addition and S for subtraction S Enter a and b values: 8 3 Subtraction is 5

b) Single character output:

The putchar( ) function is identical in description to the getchar( ) function except the following difference. putchar( ) writes a character to the stdout data stream.

putchar(char_variable_name);

Ex:
```
char ch;
Ch=getchar();
putchar(ch);
```

ii) Formatted Input and Output functions:

When input and output is required in a specified format the standard library functions scanf( ) and printf( ) are used. The scanf( ) function allows the user to input data in a specified format. It can accept data in a specified format. It can accept data of different data type. The printf( ) function allows the user to output data of different data types on the console in a specified format.

a) Formatted input:

C using scanf function for formatted input. We shall explore all of the options that are available for reading the formatted data with scanf function.

The general format is:

scnaf(―control_string‖, var1,var2,var3,…..,varn);

The control string specifies the field format in which the data is to be entered and the variables var1, var2, var3,…..,varn specify the address of locations where the data is Stored. Control string and variables separated by commas.

Ex:

```
#include<stdio.h>
main()
{
        Int a, b, c;
        printf(―\nEnter a and b values: ‖);
        scanf(―%d %d‖, &a, &b);
        c = a+b;
        printf(―\n Addition is %d‖,c);
}
```

b) Formatted output:

The printf( ) function do differe from the sort of functions that are created by the programmer as they can take a variable number of parameters.

The general format is:

printf(―control_string‖,variable1,variable2,…..,variable);

Ex:

- printf(―welcome‖);
  The above statement displays **welcome** only.
  printf(―sum is %d‖, s);
  The above format specifier %d means convert the next value to a signed decimal integer, and hence will print **sum =** and then the value passed by the variable named **s** as a decimal integer.

Aug 18

UNIT-III

SELECTION- STATEMENTS

* Sometimes it is desirable to alter the Sequential flow of control to provide for a choice of action. This Requires a logical test based on a test Expression to be carried out at some particular point with in the program.

  The action will then be carried out depending on the outcome of logical test. This is called "Condition Execution".

* Conditional-Execution, in which one group of statement is selected from several available groups, is known as "SELECTION".

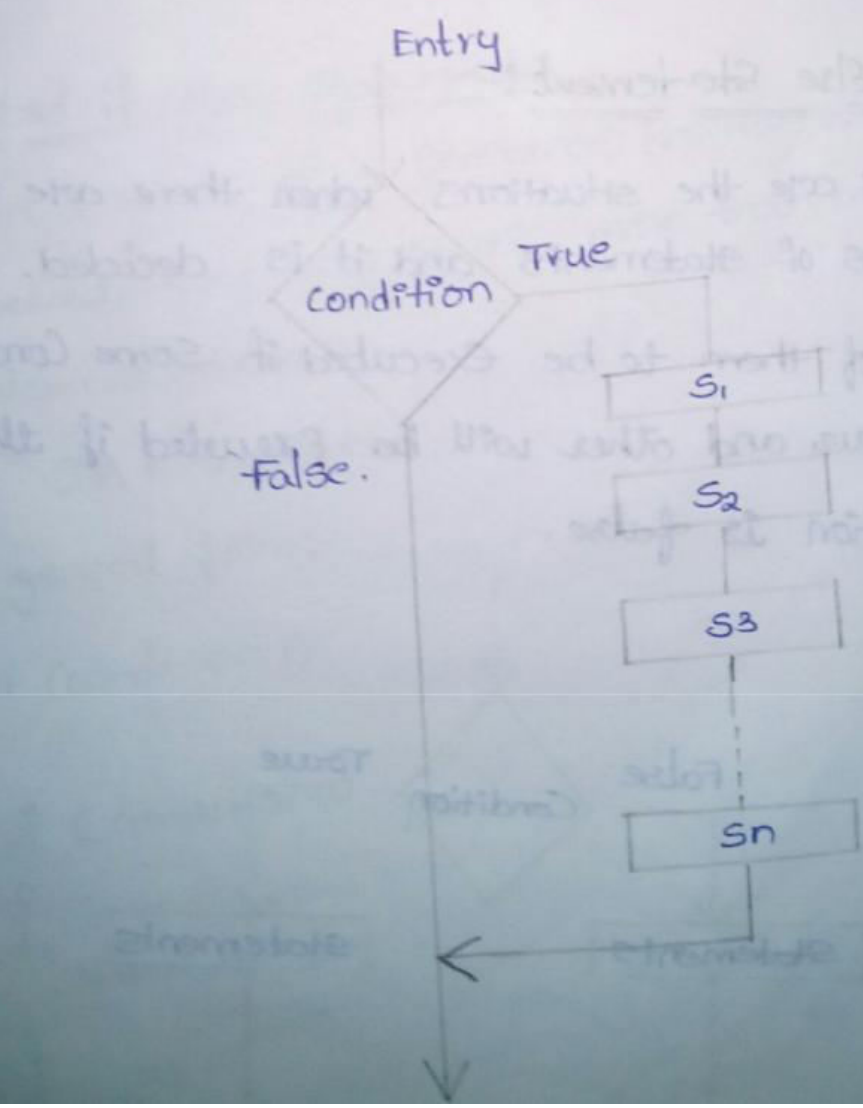
1) Simple if statement
2) if else statement
3) Nested if (Statement)-Else statement.
4) else if ladder.
5) Switch statement.
6) Ternary Statement.

* Simple-if statement:-

The if statement is used to specified Condition Executions, programm statements (or) a group of statements.

The general format is if (condition)

```
{
statement 1;
statement 2;
.
.
.
}
```



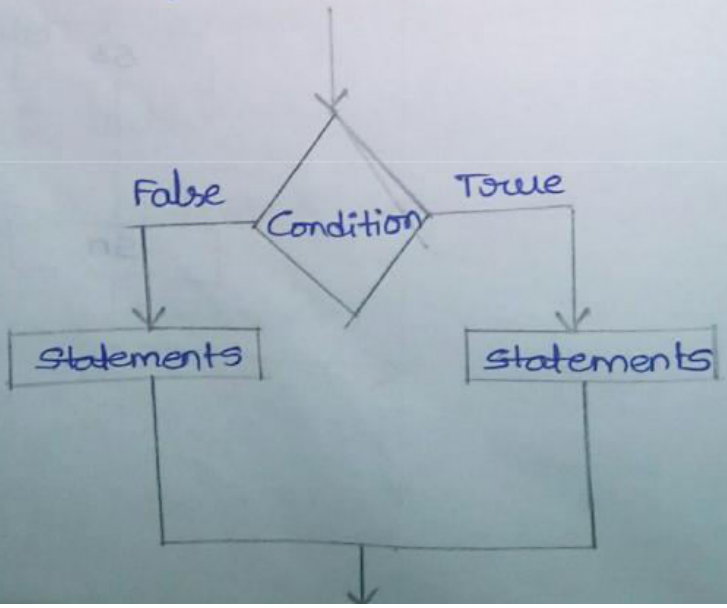
Example:-

```
main( )
{
int n;
printf ("Enter n Value");
scanf ("%d", &n);
if (n==0)
printf (" You entered zero");
}
```

Aug-19 *

If-Else Statement:-

There are the situations when there are two groups of statements and it is decided. One of them to be executed if some Condition is true and other will be executed if the Condition is false.



Eg:- /* Biggest of two Numbers */

```
main( )
{
int a,b;
printf ("Enter a,b values");
scanf ("%d %d", &a, &b);
if (a>b)
printf ("a is big");
else
printf ("b is big or Equal");
}
```

* Nested if else statement:-

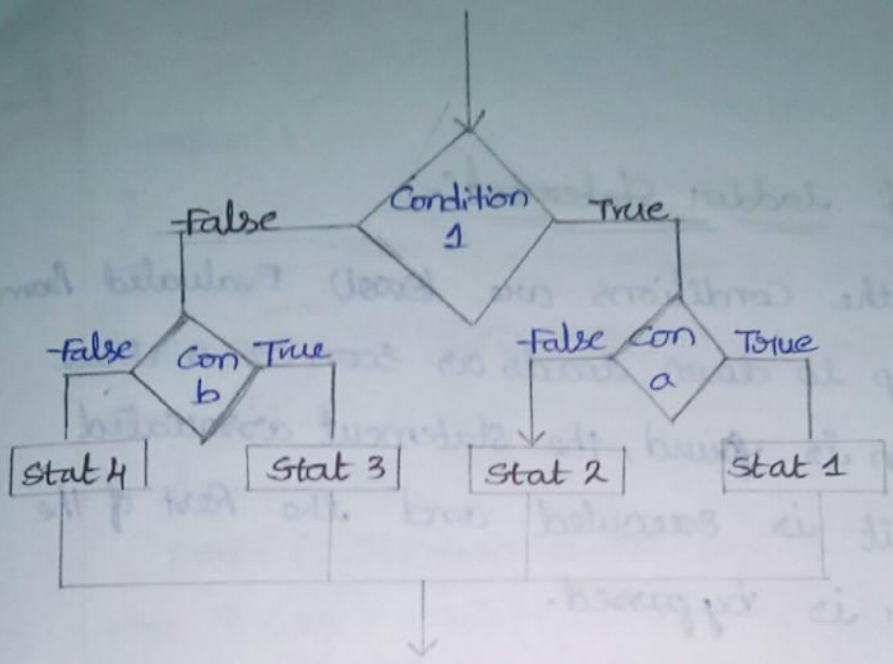
when a series of a (distance) Resistance are Involved, they may to use more than one if-Else statement. (Nested)

In Nested form

The general form:-

```
if (condition 1)
{
 if (conditional)
 {
  statement 1;
 }
 else
 {
```

```
        statement 2;
    }
}
else
{
  if (condition b)
  {
  statement 3;
  }
  else
{
  statement 4;
}
}
```



Ex:- /* Biggest of three numbers */

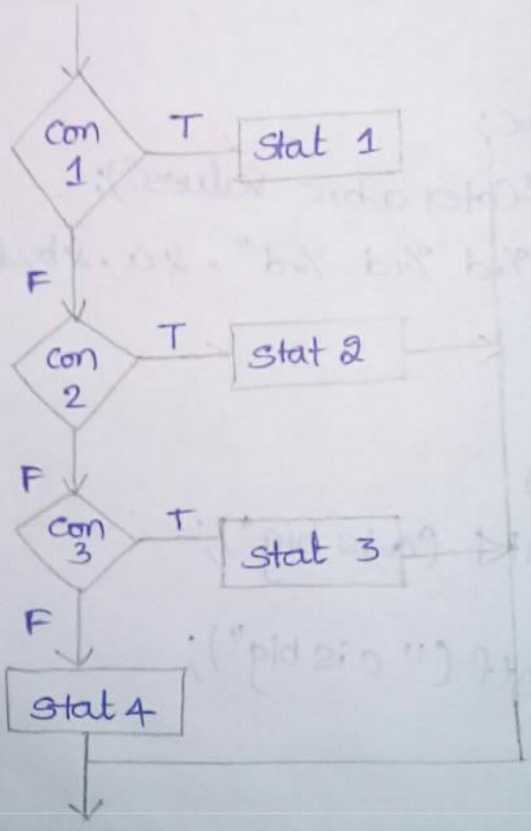
```
main( )
{
 int a,b,c;
printf("Enter a,b,c values");
Scanf("%d %d %d", &a, &b, &c);
if (a>b)
{
 if (a>c)
     printf ("a is big");
  Else
     printf ("c is big");
}
Else
{
  if (b>c)
     printf ("b is big");
   Else printf ("c is big");
}
```

9.

* **else-if ladder statement:-**

Here the conditions are (Exp) Evaluated from the top to down loads as soon as a true condition is found, the statement associated with it is executed and the Rest of the ladder is bypassed.

The last else is handles the default case.




```
if (condition 1)
    statement 1;
else if (Condition 2)
    statement 2;
else if (Condition 3)
    statement 3;
else
    statement 4;
```

Eg:

```
main( )
{
 int age ;
 printf ("Enter age Value");
scanf ( "%d", & age);
if (age >= 80)
   printf ("Sty fund is 1000 ");
else if (age >= 70)
   printf ("Styfund is 800");
else if (age >= 60)
  printf ("Styfund is 600");
else
   printf ("No Styfund");
}.
```

* Switch Statement :-

Switch statement works on same way as if-else-if but it is more eligant.

The switch statement is the special multi way decision maker that test whether ~~(decisi) (Resist)~~ * An Expression matches one of may have number of Constant values and branch according to the conditions.

```
switch (condition)
{
case value 1 : statement 1 ; break;
case value 2 : statement 2 ; break;
case value 3 : statement 3; break;
        |
        |
        |
case value n: statement n; break;
default : statement, break;
}.
```

Aug 20

Ex:-

```
main ( )
{
int n;
printf ("Enter number");
scanf ("%d", &n);
switch(n)
{
case 1 : printf ("SUNDAY"); break;
case 2 : printf ("MONDAY"); break;
      |
      |
default : printf("No day"); break;
}
{
```

```
if (n==1)
P("SUNDAY");
else if (n==2)
P("MON");
```

* A switch statement tests the value of given variable expression against a list of values.

when a Match is found a block of statements associated with the case is Executed.

In the above Example, the Expression is an Integer expression and the values are 1,2,3,4,5,6,7. The break statement is End of Each drop.

* TERNARY STATEMENT:-

'C' provides Conditional Evaluation operator is called Ternary statement.

in the form of ? :

The general format is (condition) ? Exp1 : Exp;

The ? operator relates the condition that preceeds it. if it is true, it Returns Exp 1, else it returns Exp 2.

Ex:-

```
main( )
{
int a,b;
printf ("Enter two numbers");
scanf ("%d %d", &a, &b);
c = (a>b) ? a;b;
printf (" max is %d", c);
}.
```

ITERATIVE STATEMENTS (REPEATATION)

* These statements are allow a set of Instructions to be performed until a certain condition is Reached.

'c' provides three different types of loops.

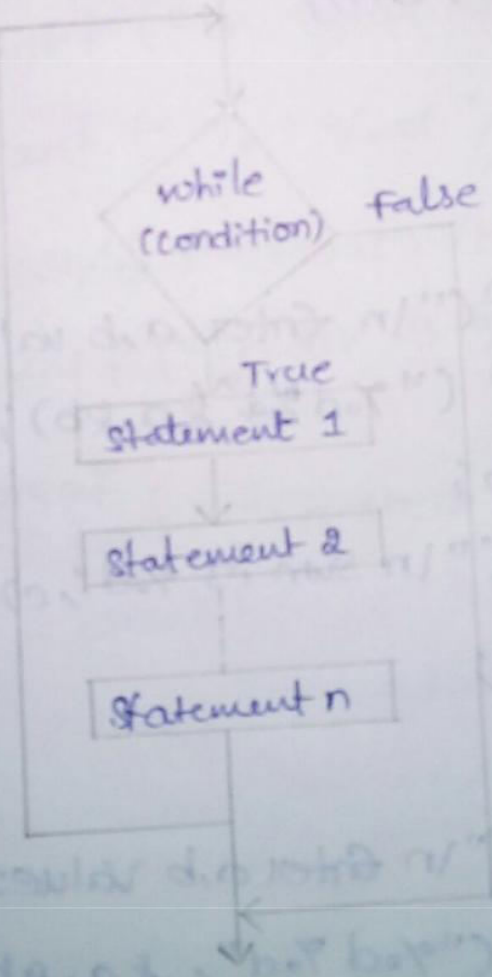
namely ;; i while loop
ii do-while loop
iii for loop

i, WHILE LOOP:-

The while loop in the 'c' starts with while key word, followed by a parameters rolling condition, and has a set of statements which Constitute the body of the loop.

The general format is;

```
while (condition)
{
 statement 1;
 statement 2;
    :
 statement n;
}
```



Switch Statement

```
#include <stdio.h>
main()
{
int a,b,c,n;
float f;
printf("\n1:add 2.sub 3.mul 4.div");
printf("\nEnter Number");
scanf("%d",&n);
switch(n)
{
case 1;
        printf("\n Enter a,b values");
        scanf("%d%d",&a,&b);
        c=a+b;
        printf("\n Sum is %d",c);
        break;
case 2;
        printf("\n Enter a,b values");
        scanf("%d%d",&a,&b);
        c=a-b;
        printf("\n sub is %d",c);
        break;
```

```
case 3 :
        printf ("\n enter a,b values");
        scanf ("%d %d", &a, &b);
        c = a * b;
        printf ("\n Mul is %d", c);
        break;

case 4 :
        printf ("\n enter a,b values");
        scanf ("%d %d", &a, &b);
        if (c=a/b) f=(float)a/b;
        printf ("\n div is %f", f);
        break;
default : printf ("\n there is no operation");
        break;
}
}.
```

Aug 26

* As soon As Execution reaches while loop, the Specified Condition is tested if it is form to be true, it will Enter into the body of the loop. Once it Reaches the closing ~~Rays~~ trace of the body. automatically loop back to the top and test the Condition forcely now And if it is true re-Enter the body and so on. till the Controlling Condition of the loop becomes false.

Eg:-

```
while (condition)
{
  =
}.
```

* Factorial *

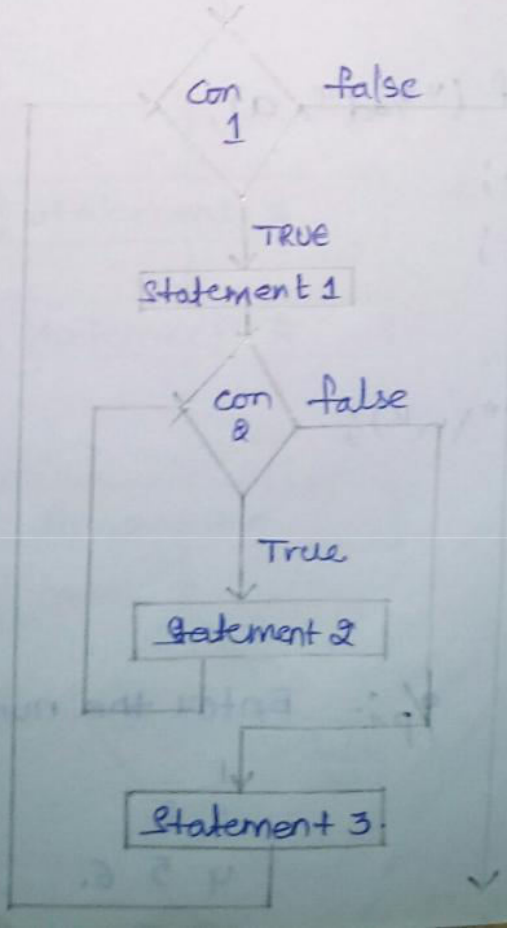
```
#include <stdio.h>
# include <conio.h>
main ( )
{
int n,i=1,f=1;
clrscr ( );
printf (" \n Enter n value");
Scanf ("%d", &n);
```

```
while (i<=n)
{
f= f * i;
i= i+1;
}
printf ("\n factorial is %d", f);
getch();
}.
```

Nested Loop's

b) when a while loop is Executed, instead of there is one or more Internal loops. Those loops are called "Nested loops".

The general format is



```
/* floyd's triangle */

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

o/p:- Enter the number 3

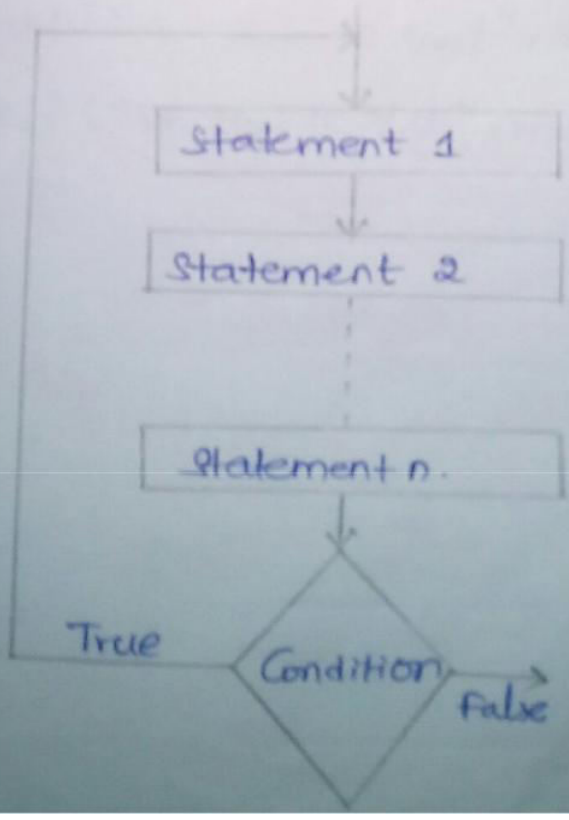
1
2 3
4 5 6.

Do-while loop:-

The Do while loop performes the test at the bottom Rather than at the top. The Do while loop starts with key word DO, followed by the body of the loop.

The general format is

```
do {
    Statement 1;
    Statement 2;
    :
    Statement n;
} while(Condition);
```



*Eg:-

```
main( )
{
int n,i=1, f=1;
printf ("Enter n value");
Scanf ("%d", &n);
do {
f = f * i;
i = i+1;
} while (i<=n)
printf ("factorial is %d", f);
}.
```

Aug 29

Add Digits & Multiplication of a Number

Eg:-

```
main( )
{
 int n, Sum = 0, mul = 1, r;
printf ("enter Number ");
Scanf ("%d", &n);
while (n!=0)
{
  r = n % 10;
  n = n/10;
  Sum = Sum + r;
  mul = mul * r;
}
printf ("\n Sum is %d", Sum);
printf ("\n Mul is %d", mul);
{.
```

Arm Strong Number:

```
*  main()
{
 int n,r,arm=0,temp;
 printf("Enter number");
 scanf("%d",&n);
 temp=n;
 while(n!=0)
 {
   r=n%10;
   n=n/10;
   arm=arm+(r*r*r);
 }
 if(temp==arm)
   printf("\n Arm Strong");
 else
   printf("\n Not arm strong");
}
```

Aug 31

* (c) For loop:-

This is used when the statements are to be Executed more than once.

This is the most quitely used iteration loop.

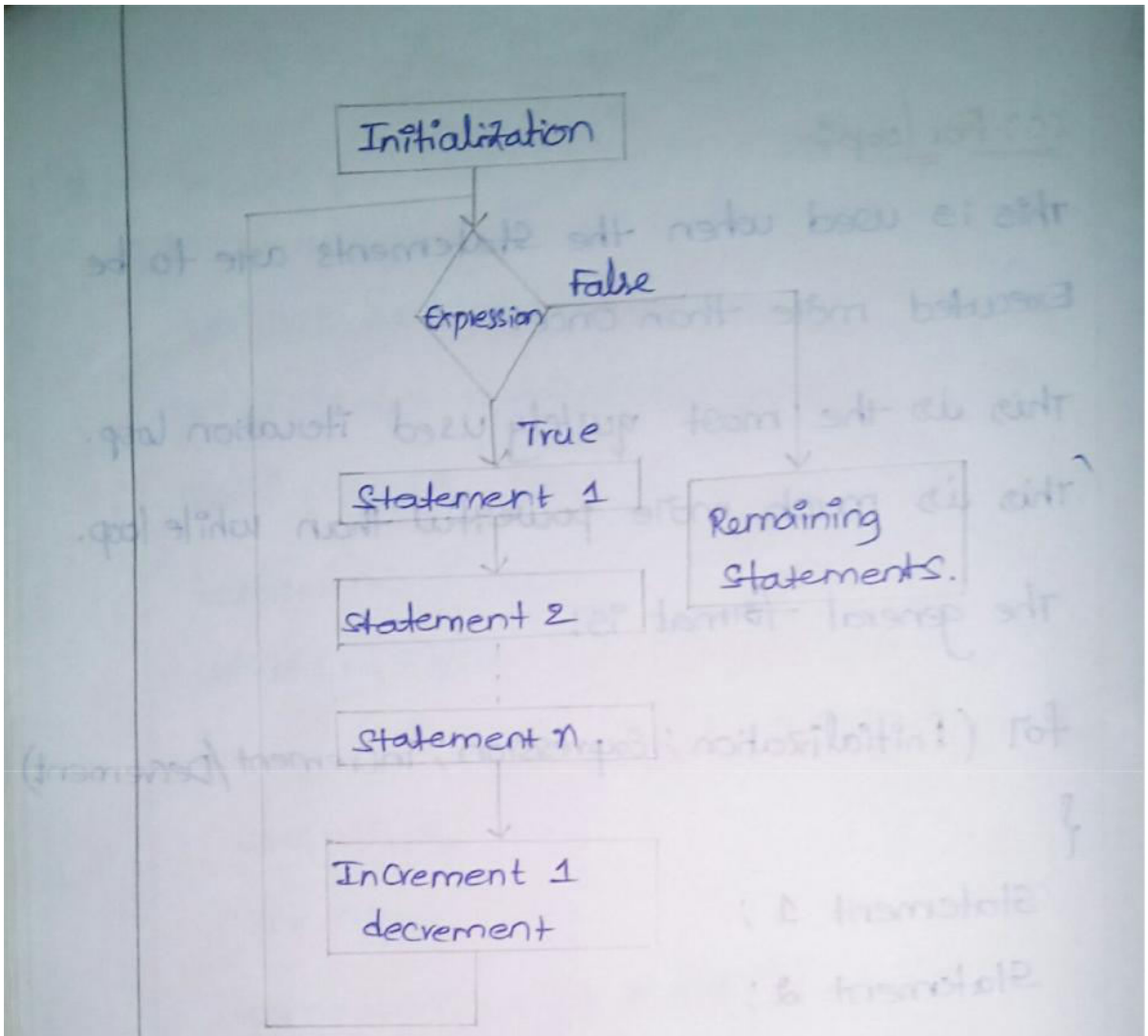
This is much more powerful than while loop.

The general format is:-

```
for (Initialization ; Expression ; inCrement/Decrement)
{
  Statement 1 ;
  Statement 2 ;
        :
        :
  Statement n ;
}
Remaining statements;
```

Eg:

```
main( )
{
int n,i,f=1;
printf ("Enter Number");
scanf ("%d", &n);
for (i=1; i<=n; i++)
{
 f = f*i;
}
printf ("\n fact is %D", f);
}
```



Eg:-

```
main( )
{
int n, i = 1;
printf ("Enter number");
Scanf ("%d", &n);
for (i<=10) i<=n; i++)
{
printf ("\n %d * %d = %d", n, i, n*i);
}

}
```

```
main()
{
int n,i,k;
clrscr();
printf("Enter n value");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(k=1;k<=i;k++)
{
printf("%d",k);
}
printf("\n");
}
}
```

Sep-01

JUMP STATEMENTS

* (i) break
  (ii) Continue
  (iii) go to

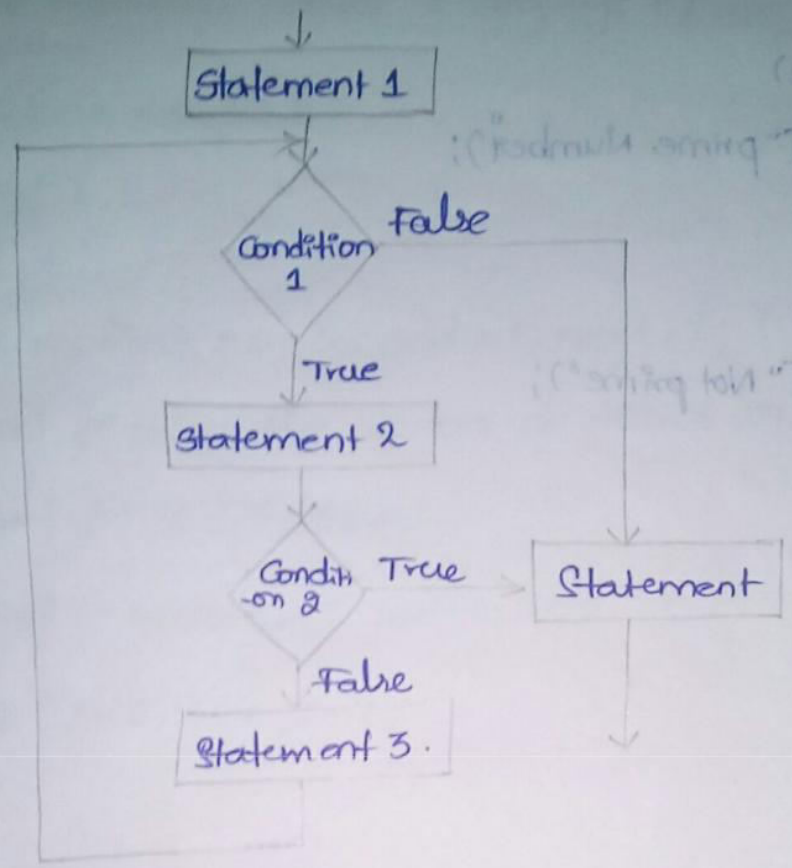
* (i) break;-

In 'c' programming the break statement can break the inner most control structure. we already have used the break statement in switch statement and also use inside a while loop, a for loop and do-while loop.

The general format is

```
Statements 1;
while (Condition)
{
 Statement 2;
 if (Condition 2)
  break;
 Statement 3;
}
 Statement 4;
```

Eg:



Eg: For loop ⇒ prime Number

```
#include <stdio.h>
main( )
{
int n, i=1, f=0;
printf ("Enter the number");
scanf ("%d", &n);
for (i=1 ; i<=n ; i++)
{
 if (n%i==0)
{
f=f+1;
}
```

```
}
if (f==2)
printf ("prime Number");
else
{
printf ("Not prime");
 }
}
```

```
/* Fibonacci Series c language */

#include <stdio.h>
main( )
{
int n, first = 0, second = 1, next, i;
printf ("Enter the number of terms \n");
scanf ("%d", &n);
printf ("Fibonacci series are", n);
for (i=0; i<n; i++)
{
if (i<=1)
   next = i;
   else
{
  next = first + second;
  first = second;
  second = next;
  }
  printf ("%d", next);
  }
}
```

output

Enter the number of terms 5

fibonacci Series are 0 1 1 2 3

- 8

- 0 1 1 2 3 5 8 13

- 12

- 0 1 1 2 3 5 8 13 21 34 55 89

2. pascal

```
#include<stdio.h>
long fact (int);
main()
{
int i,n,c;
printf("Enter the number of rows");
Scanf ("%d", &n);
for ( i=0; i<n ; i++)
{
for ( c=0; c<=(n-i-2); c++)
Printf (" ");  for (c=0,c<=i; c++)
print f ("%4d", fact (i)/fact (c) * fact (i-c)));
print f ("\n")
}
}
long fact (int n)
{
int c;
long result = 1;
for (c=1; c<=n; c++)
```

```
result = result * c;
return result;
}
```

output

Sep-20

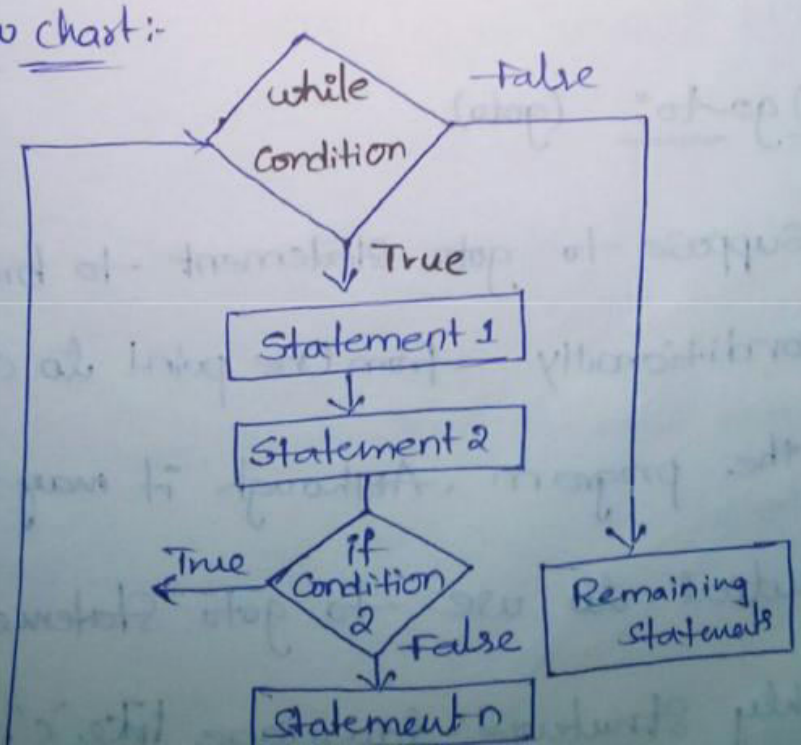
* (11) Continue :-

The Continue statement, whenever Executed the Rest of current Iteration to be scipped and causes the next iteration to begin.

The general format is,

```
while (condition)
{
    statement 1;
    statement 2;
    if (condition 2)
        continue;
    statement n;
}
```

Flow chart :-



* (go to:)

Example:

①
```
#include <stdio.h>
main( )
{
int i;
while (i<=10)
{
printf ("i value is %d", i);
if (i==5)
Continue;
i++;
}
}
```

②
```
main( )
{
int i=0;
while (i<=10)
{
i++;
if (i==5)
continue;
printf ("i value is %d", i);
}
}
```

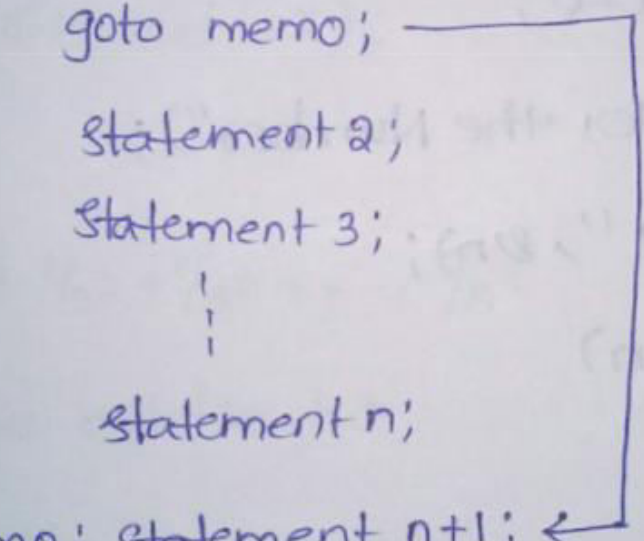
(iii) goto:- (goto).

c Suppose to goto statement to branch unconditionally from one point to another point in the program. Although it may not be Essential to use to goto statement in a highly structure language like 'c'.

The goto Requires a variable in order to identify its place where the branch is to be made.

The general format :-

```
        statement 1;
        goto memo;
        statement 2;
        statement 3;
           :
           :
        statement n;
memo:   statement n+1;
        statement n+2;
        statement n+3;
```



1. $x = 1+2+3+4+---+n$

```
#include <stdio.h>
#include <conio.h>
main()
{
int i=1, n, x=0;
printf ("Enter the Number");
scanf ("%d", &n);
while (i<=n)
 {
 x=x+1;
  i++;
}
printf ("x is %d", x)
}
```

2. $x = 1^2+2^2+3^2+ -- +n^2$

```
#include <stdio.h>
main()
{
int i=1, n, x=0;
printf ("Enter the number");
```

```
scanf("%d", &n)
while (i<=n)
 {
 x=x+i*i)
   i++;
}
printf ("x is %d", x);
}
```

3. $x = 1/1^2 + 1/2^2 + 1/3^2 + \cdots 1/n^2$

```
#include <stdio.h>
main( )
{
int i=1, n, x=0;
                  → float x=0.0;
printf ("Enter the number");
scanf ("%d", &n);
while (i<=n)
{
x = x+1/(i*i);
   i++;
}
printf ("x is %d", x)
                   %f
}.
```

UNIT-4

FUNCTIONS

A C-language program is nothing but collection of Function; these are the building blocks of a „C‟ program. Generally, a function mans a task.

“Function is a collection of logically related instructions which performs a particular task.”

Functions are also used for modular programming in which a big task is divided into small modules and all the modules are inter connected.

How to create the function (Function declaration):-

<return value type><function name> (parameter list)
{
Body of the function;
[return<value>];
}
<return value type>: It is specify the data type of the returning value.
<function name>:It is specify the name of the function.
(Parameter list): It is the list of parameters used in the function.

Function definition:-

The collection of program statements in C that describes the specific task done by the function is called a Function definition. It consists of the function header and a function body.

<return type><function name>(parameter list)

{
<local variable declaration>;
<Statements>;
<return (value)>;
}

Function header:

Function header is similar to the function declaration but does not require the semicolon at the end. List of variables in the parenthesis are called Formal parameters. It consists of three parts:

1. Return type.
2. Function name.
3. Formal parameters.

Example:

int add(a, b)

Function body:

After the function header the statements in the function body (including variables and statements) called body of the function.

Example:

```
int add (int x, int y)
{
return (x+y);
}
```

Return statement:

Return statement is used return some value given by the executable code within the function body to the point from where the call was made. General form:

```
return expression;
      or
return value;
```

Where expression must evaluate to a value of the type specified in the function header for the return value.

EX: return a+b;

Or

return a;

If the type of return value has been specified as void, there must be no expression appearing in the return statement it must be written simply as

```
return;
```

Function call (Function calling):

All functions within standard library or user-written, are called with in this main() function, the function call statements involves the function, which means the program control passes to that of the function. Once the function completes its task, the program control is passed back to the calling environment.

Syntax:

<function name><(parameter list)>;

or

variable name=<function name><(parameter list)>;

Example:

1.Write a program to find the mean of the two integer numbers.

```
#include<stdio.h>
int mean(int, int);
main()
{
int p, q, m;
clrscr();
printf("enter p,q values");
scanf("%d%d",&p,&q);
```

```
m=mean(p,q);
printf("mean value is %d",m);
getch();
}

int mean(int x, int y)
{
int temp;
temp=(x+y)/2;
return temp;
}
```

2. Write a program that uses a function to check whether a given year is leap or not.

```
#include<stdio.h>
void leap(int);
main()
{
int  year;
clrscr();
printf("\n enter year \n");
scanf("%d",&year);
leap(year);
}
void leap(int yr)
{
if(year%4==0&&year%100!=0||year%400==0)
printf("leap year");
else
printf("not leap year");
}
```

Types of functions:

The functions are also used for modular programming in which a big task is divided into small modules and all the modules are interconnected. Functions are basically classified as:

1. Standard library functions(built-in functions).
2. user defined functions.

Standard library functions:-

Library functions come along with the compiler and they can be used in any program directly. User can"t be modified the library functions.

Example:

sqrt();, printf();, clrscr();, abs();, strlen();, getch();etc……

Example program:

```
#include<math.h>
main()
{
int x;
printf("enter x value \n");
```

```
scanf("%d",&x);
printf("square root value of x is %d", sqrt(x));
getch();
}
```

user defined functions: user defined functions is one which will be defined by the ser in program to group certain statements together. The scope of a user defined function is to the extent of the program only, in which it has been defined. main is also a user defined function.

Syntax:

```
<returntype><function name)(<parameter list>)
{
Function body;
<return value>;
}
```

Example program:

```
#include<stdio.h>
main()
{
int r,rd;
clrscr();
printf("enter radius for circle\n");
scanf("%d",&r);
rd=area(r);
printf("area of the circle is %d",rd);
getch();
}
area(int x)
{
return(3.14*x*x);
}
```

Types of parameters

The nature of data communication between the calling function and the called function with arguments (parameter).

The parameters are two types:

1. Actual parameters or original parameters.
2. formal parameters or duplicate parameters.

```
main()
{
 ……
 Function1(a1, a2, a3,……am);
 ……
}

 Function1(x1, x2, x3,……xn);
{
```

```
  …….
  ……....
  …….
}
```

Actual parameters(arguments):

Actual parameters means original parameters for the function call. These parameters are declared at the time of function declaration. In the above example a_1,a_2,a_3,…… a_m are the actual parameters. When a function call is made, only a copy of the values of actual arguments is passed into the called function.

Formal parameters (duplicate arguments):

Formal parameters mean duplicate parameters for function call. These parameters are declared at the time of function definition. In the above example x_1, x_2, x_3,……x_n are the formal parameters, the actual and formal arguments should match in the number, type and order, the values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.

Types of functions based on parameters:

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:

Function which does not have any argument, receive no data from the calling function and not return any value to calling function. Example program:

```
#include<stdio.h>
main()
{
int ch;
printf("1. addition");
printf("2. substraction");
printf("3. multiplication");
printf("enter ur choice from (1-3)");
scanf("%d",&ch);
if(ch>3)
error();
getch();
}
error()
{
```

```
printf("ur choice is wrong");
}
```

Functions with arguments and no return value:

This is another type where one way communication is possible between the callings and called function. That is, the called function will receive data from the calling function, but will not transfer any data to it.

Example program:

```
#include<stdio.h>
main()
{
int a,b,c;
printf("enter 3 numbers");
scanf("%d%d%d",&a,&b,&c);
big(a,b,c);
}
big(x,y,z)
int x,y,z;
{
if(x>y&&x>z)
printf("\n %d is biggest",x);
else if(y>z)
printf("\n %d is biggest",y);
else
printf("\n %d is biggest",z);
}
```

Functions with arguments and return value:

In the type, two way data communication takes place. That is, both the called and calling functions receive and transfer data from each other. Example program:

```
#include<stdio.h>
main()
{
float x,y,z,area();
printf("enter base and height");
scanf("%f%f",&x,&y);
c=area(x,y);
printf("the area is %f sq.units",c);
getch();
}
float area(float b, float h)
{
return(5*b*h);
}
```

Functions with no arguments and return value.

Function which does not have any argument, receive no data from the calling function and return a value to calling function.

Example program:

```
#include<stdio.h>
main()
{
int x;
clrscr();
x=square();
printf("square of 3 is %d",x);
getch();
}
square()
{
return(3*3);
}
```

Nested functions:-

One important advantage of c functions is that they can be called from and within another function. All the called and calling functions transfer and receive data with ach other and this is called "Nested function".

Example program:

```
#include<stdio.h>
main()
{
printf("\n I am in main");
fun1();
printf("\n again I am in main");
getch();
}
fun1()
{
printf("\n I am in function1");
fun2();
}
fun2()
{
printf("\n I am in function2");
}
```

Function prototype:

Function prototype means a function have return type, function name, parameter list and must be end with semicolon(;),

Example program:

```
#include<math.h>
#include<stdio.h>
int squareroot(int);      ---→ function prototype
main()
{
```

```
int m,n;
clrscr();
printf("enter n value \n");
scanf("%d",&n):
m=squareroot(n);
printf("the squareroot of n is %d",m);
getch();
}
int squareroot(int x)
{
return(sqrt(x));
}
```

Storage classes:

The variables are declared by the type of data they can hold. During the execution of the program, these variables may be stored in the registers of the CPU or the primary memory of the computer. C provides four storage class specifier they are:

1. automatic.
2. external.
3. registers.
4. static.

The storage class specifier precedes the declaration statement for a variable.

Syntax:

<storageclass specifier><data type><variable name>;

Automatic storage class:

By default all variables declared with in the body of any function are automatic. The keyword "auto" is used in the declaration of a variable to explicitly specify its storage class.

Ex:

```
auto int a=10;
```

"a" is a variable that can hold a integer and its storage class is automatic. Even if the variable declaration statement in the function body does not include the keyword "auto", such declared variables are implicitly specified as belonging to the automatic storage class.

Example program:

```
#include<stdio.h>
int i=10,x=20;
main()
{
auto int i=20;
printf("the I value is %d",i);
printf("\n x value is %d",x);
getch();
}
```

Automatic storage class variable scope starts when it is created and the scope ends when the function is closed.

Storage area: RAM

Default value: garbage value

Scope: local to the block in which the variable is defined.

Life time: till the control remains with in the block in which the variable is defined.

External storage class:

Usually a big program is divided into a number of small programs, so that maintenance of the program becomes easy. A variable defined in a program can be accessed by another program as well, such a variable is called as extern (external) variable.

The keyword for declaring such global variable is extern.

Syntax:

```
extern datatype variablename;
```

Example program-1:

```
#include<stdio.h>
int i=10;
main()
{
int i=20;
printf(" %d",i);
incre();
}
incre()
{
printf("\n%d",i);
getch();
}
```

A program can have same variable name as global variable and local variable. In such case the local will have precedence over the global variable.

Storage area: RAM

Default value: zero

Scope: global.

Life time: external.

Example program-2:

```
extern int i=10;
main()
{
clrscr();
printf("\n %d",i);
incr();
getch():
}
incr()
{
printf("\n %d",++i);
}
```

output:

1 0
1 1

Static storage class:

The static storage class used for fixed storage for variable within a specified region. Two kinds of variables are allowed to be specified as static variables: static variables and local variables. The local variables are also referred to as internal variables while the global variables are also known as external variables. The default value of static variables is zero.

Storage area: RAM
Default value: zero
Scope: local.
Life time: internal.

Syantax:

```
static int a=10;
```

Example program-1:

```
#include<stdio.h>
main()
{
clrscr();
printf("\n first class\t ");
show();
printf("\n second class \t");
show():
printf("third class \t");
show();
getch();
}
show()
{
static int i;
printf("%d",i);
i++;
}
```

Example program-2:

```
#include<stdio.h>
main()
{
int x;
clrscr();
printf("factorial of 1-7 \n");
printf(" %d\n",show(x));
}
getch();
}
show(int z)
```

```
{
static int i=1,fact=1;
for( ;i<=z;i++)
fact=fact*i;
return fact;
}
```

Registers storage class variables:

Variables stored in the registers of the CPU are accessed in much lesser time than those stored in the primary memory. The keyword this storage class is "registers", this keyword precedes the normal declaration statement of a variable. Example program:

```
#include<stdio.h>
int power(int,int)
main()
{
int num=4;
int k=3;
int x;
x=power(num,k);
printf("\n value of num=%d, raised to k=%d is %d",num,k,x);
}
int power(int p,rgister int q)
{
register int temp;
temp=1;
int i;
for(i=1;i<=q;i++)
temp=temp*i;
return temp;
}
```

Storage area: Registers
Default value: garbage value
Scope: local.
Life time: internal.

Scope rules:-

The scope relatives to the accessibility, the period of existence, and the boundary of usage of variables declared in a statement block or function. If any variables declared inside the block, that variables scope is in the block only. If any variables declared in the main(), that variable scope is with in the main and their sub blocks only. If any variable declared outside the main(). As global, that variable can accessed entire program means, main() and their sub program.

Block structure:

Block structure means it is the blocks of executable statements, constructed with curly braces({}). The block of statements enclosed with open curly brace and closed curly brace. The scope rules in the block of C code, lying with in curly braces({}),

basically specifies the accessibility duration of existence and boundary of usage of the variable.

Example program: for scope rules and block structure.

```
#include<stdio.h>
main()
{
int x=3;
printf("\n x value in outer block %d",x);
{
int x=45;
printf("\n x value in inner block %d",x);
}
printf("\n again x value in outer block %d",x);
getch();
}
```

Recursion:

A recursion function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self call.

Need of recursion

1. decomposition into smaller problems of same type
2. recursive calls must reduce problem size
3. a recursive function would call itself indefinitely
4. recursion is like a top-down approach

Factorial of a number:

n!=n*(n-1)*(n-2)*……..*1 for n>0

0!=1

Example program:

```
#include<stdio.h>
main()
{
int m,n;
clrscr();
printf("enter n value");
scanf("%d",&n);
m=fact(n);
printf("factorial of n is %d",m);
getch();
}
fact(int x)
{
if(x==1)
return 1;
else
return(x*(fact(x-1));
}
```

Recursion for fibonacci sequence:

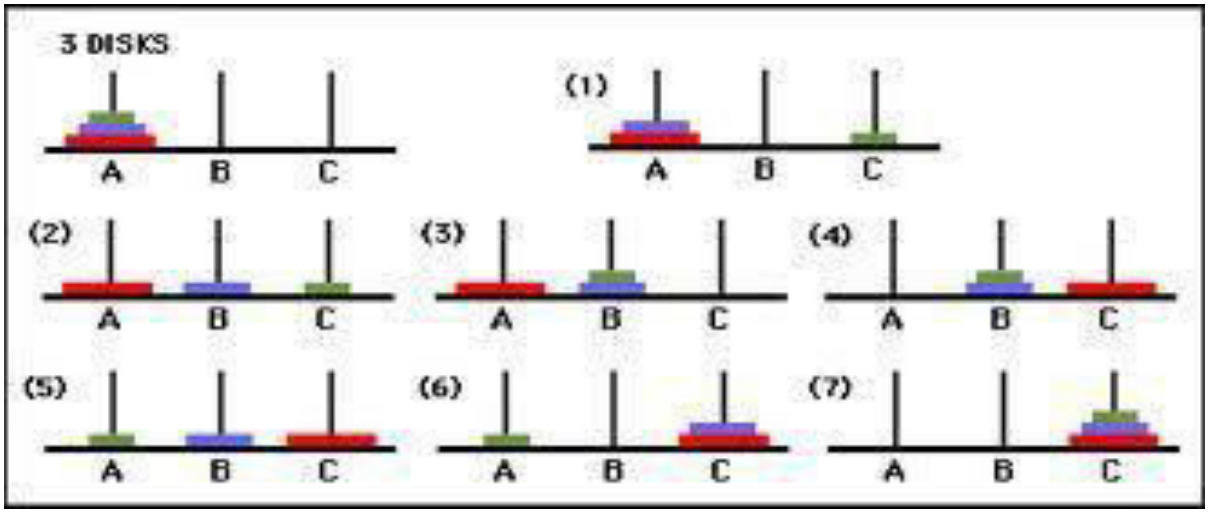
```
#include<stdio.h>
int fib(int val);
main()
{
int i,j;
clrscr();
printf("\n enter th number of terms");
scanf("%d",&i);
printf("Fibonacci sequence for %d terms is:",i);
for(j=0;j<=i;j++)
printf("%d",fib[j]);
getch();
}
int fib(int val)
{
if(val<=2)
return 1;
else
` return(fib(val-1)+fib(val-2));
}
```

The towers of Hanoi:

The towers of Hanoi problem is a classic case study in recursion, it involves moving a specified number if disks from one tower to another using a third as an auxiliary tower.

Specification move n disks from peg A to peg C, using peg B as needed.

- Only one disk may be moved at a time.
- This disk must be the top disk on a peg.
- A larger disk can never b placed on top of a smaller disk.



Towers of Hanoi

The problem is not to focus on the first step, but on the hardest step i.e, moving the bottom disk to peg C.

- move disk3 from peg A to peg C
- move disk2 from pg A to peg B
- move disk3 from peg C to peg B
- moving disk1 from peg A to peg C
- moving disk3 from peg B to peg A
- moving disk2 from peg B to peg C
- moving disk3 from peg A to peg C

Parameter passing:

Parameter passing from calling function to called function is in two ways. One is call by value, second is call by reference.

Call by value:

While calling a function, the values in the arguments are passed by value to the formal parameters of the function. In fact, only copies of the values held in the arguments are sent to the formal parameters.

Example program:

```
#include<stdio.h>
main()
{
int a,b;
clrscr();
printf("enter a,b values");
scanf("%d%d",&a,&b);
printf("before swapping a=%d, b=%d",a,b);
swap(a,b);
getch();
}
swap(int x, int y)
{
int c;
c=x;
x=y;
y=c;
printf("after swapping a=%d, b=%d",x,y);
}
```

Call by reference:

In another function call technique known as "call by reference", more strictly termed as "call by address" where values are passed by handling over the address of arguments to the called function.

Example program:

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

```
{
int a,b;
clrscr();
printf("enter a,b values");
scanf("%d%d",&a,&b);
printf("before swapping a=%d, b=%d",a,b);
swap(&a,&b);
getch();
}
swap(int *x, int *y)
{
int *c;
*c=*x;
*x=*y;
*y=*c;
printf("after swapping a=%d, b=%d",*x,*y);
}
```

C pre-processor:

C pre-processor is also called a macro processor. A macro is defined as an open-ended subroutine the preprocessor provides its own language that can be a very powerful tool for the programmer. These tools are instructions to the preprocessor and are called directions. The **C Preprocessor** is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

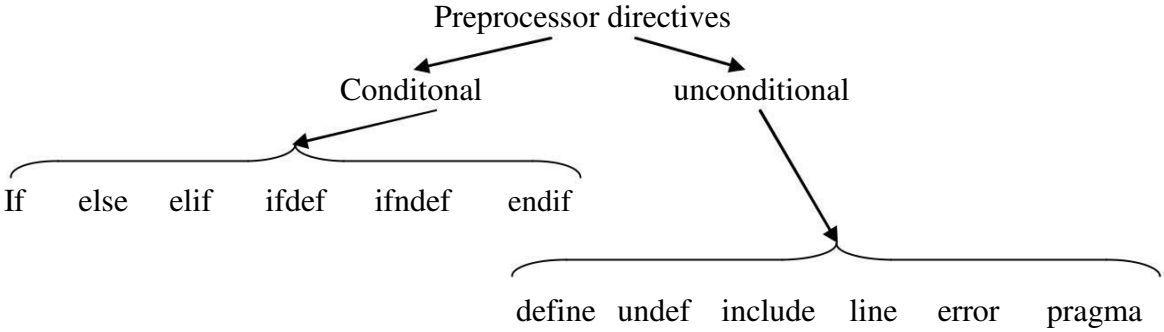
All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Advantages:

1. Program development easier.
2. Program easier to read.
3. Modification of programs easier.
4. Transferable code.

Types of c preprocessor directives:

Pre-processor directives can be classified into two categories: unconditional and conditional.



Conditonal:

#if:

here #if is a conditional directive of the preprocessor

Syntax:

```
#if <expression>
<statement>
#endif
```

Ex:

```
#if a>0
printf("a is positive");
#endif
```

#else:

The #else is also usd with this directive if required. #if and #else pair operates in a away similar to the if-else.

Syntax:

```
#if <expression>
<statement>
#else
<statement>
#endif
```

Ex:

```
#if a>b
printf(" a is big");
#else
printf(" b is big");
#endif
```

#elif:

Syntax:

```
#if <expression>
<statement>
#elif<expression>
<statement>
#else
<statement>
#endif
```

Ex:

```
#if a>b&&a>c
printf(" a is big");
#elif b>c
printf(" b is big");
#else
```

```
printf(" c is big");
#endif
```

#ifdef and #ifndef:

The #ifdef directive executes a statement sequence if the macro-name is defined, if the macro-name is not defined, the #ifdef directive executes a statement sequence.

#ifdef:

Syntax:

```
#ifdef macroname
<statement sequence>
#endif
```

Ex:

```
#define a 1
#define b 0
main()
{
#ifdef a
printf(" a is true");
#endif
#ifdef b
printf(" b is false");
#endif
```

#ifndef:

Syntax:

```
#ifndef macroname
<statement sequence>
#endif
```

Ex:

```
#define a 1
#undef b 0
main()
{
#ifndef a
printf(" a is defined");
#endif
#ifndef b
printf(" b is not defined");
#endif
```

Unconditional:

#define:

#define directive is used to make substitutions throughout the program in which it is lacated.

Syntax:

```
#define macro-name replace-string
```

Ex:

```
#define max 10
main()
{
#if max>0
printf("condition is true");
#endif
 }
```

#undef:

This directive undefines a macro. A macro must be undefined before being redefined to different value. Undef takes single argument. Syntax:

```
#undef macro_name
```

Ex:

```
#undef value
#define value 10
#define max 20
main()
{
#if max>value
printf("condition is tru");
#endif
}
```

#include:

This directive include the files into present working directory.

Syntax:

```
#include<filename>
```

Ex:

```
void display()
{
int a=10;
printf("a value is %d",a);
}
```

This file saved as disp.c

```
#include<disp.c>
main()
{
clrscr();
display();
getch();
}
```

#error:

The directive #error is used for rporting errors by the preprocessors.

Syntax:

```
#error error_mssage
```

#line:

This directive is used to change the value of the line and file variables.

Syntax:

#line line_number <filename>

Ex: #line 20 disp.c

#pragma:

The #pragma directive is implementation specific, uses vary from compiler to compiler.

Header file creation:

In computer programming, a **header file** is a file that allows programmers to separate certain elements of a program's source code into reusable files. Header files commonly contain forward declarations subroutines, variables, and other identifiers. Programmers who wish to declare standardized identifiers in more than one source file can place such identifiers in a single header file, which other code can then include whenever the header contents are required. A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

Header file contains no.of functions, these functions are frequently used in all programs, user can also create the own header files.

Creation of header file:

```
int fact(int n)
{
int f=1,i=1;
if(n>7)
{
printf(“sorry! Can‟t find a factorial for a number>7”);
return -1;
}
else
{
for(i=1;i<=n;i++)
{
f=f*I;
}
return f;
}
}
int rverse(int n)
{
int r,rv=0;
while(n>0)
{
r=n%10;
rev=rv*10+r;
```

```
n=n/10;
}
return rev;
}
int evenodd(int n)
{
if(n%2==0)
printf("even");
else
printf("odd");
}
void Armstrong(int n)
{
int r,arm=0,n1;
n1=n;
while(n>0)
{
r=n%10;
arm=arm+(r*r*r);
n=n/10;
}
if(n1==arm)
printf("Armstrong");
else
printf("not Armstrong"):
}
```

Save the above file as prasad.h (we cant execute this file because main() is absent)

```
#include<stdio.h>
#include<prasad.h>   →use of header file prasad.h
main()
{
int n1,n2,ch;
clrscr();
printf("enter n1 value");
scanf("%d",&n1);
printf("1.factorial\n 2.reverse\n 3.evenodd\n 4.armstrong");
printf("enter ur choice from above menu");
scanf("%d",&ch);
switch(ch)
{
case 1: n2=fact(n1);
            if(n2!=-1)
            printf("factorial n1 is %d", n2);
            break;
case 2: n2=reverse(n1);
        printf("revrse no.of n1 is %d",n2);
```

```
                break;
        case 3: evenodd(n1);
                brak;
        case 4: armstrong(n1);
                break;
        default: printf("your choice is wrong");
        }
        }
```

Passing 1-D arrays, 2-D arrays to functions:

Passing arrays to functions:

Arrays can also be arguments of functions. When an entire array is an argument of a function, only the address of the array is passed and not the copy of the complete array. Therefore, when functions is called with the name of the array as the argument.

Passing 1-D array:

Syntax:

<return type><function name>(default arrayname[size]);

Example:

```
#include<stdio.h>
int maximum(int []);
main()
{
int values[5],i,max;
printf("enter elements \n");
for(i=0;i<5;i++)
scanf("%d",& values[i]);
max=maximum(values);
printf("maximum element is %d\n",max);
getch();
}
int maximum(int b[])
{
int max=0,i;
for(i=0;i<5;i++)
{
if(b[i]>max)
max=b[i];
}
return max;
}
```

Passing 2-D array to functions:

Syntax:

<returntype><function name>(datatype arrayname[size1][size2]);

Example:

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

```
{
int a[2][2],i,j,s;
clrscr();
printf("enter array elements \n");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
scanf("%d",&a[i][j]);
}
}
s=sum(a);
printf("sum is %d\n",s);
getch();
}
sum(int b[2][2])
{
int i,j,s;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
s=s+b[i][j];
}
}
return s;
}
```

UNIT-5

Array

Array: Array is a user defined datatype. Array is group of elements, these elements are homogeneous. Each element shared common name elements are variable with their index value.

Syntax: datatype arrayname[size];

Use of arrays:

1. Storing more than one value at a time under a single name.
2. Reading, processing and displaying the array elements is easy.
3. Some logics can be implemented only through arrays.

Declaration of arrays:

Syntax: datatype arrayname[size];

Example: int a[10];

Declaration of arrays with datatype, arrayname and size of the array. Size denotes the maximum number of elements that can be stored in the array. Accessing elements (initialization of array elements)

Access the array elements in two ways:

1. Design time initialization
2. Run time initialization

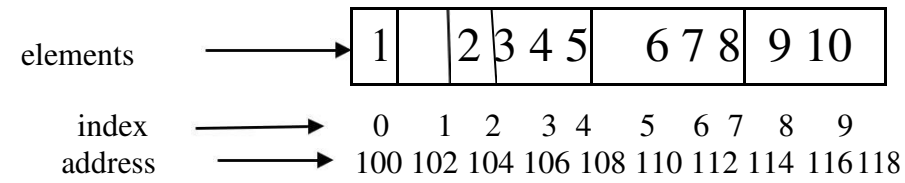
Design time accessing:

Declaration and initialization at a time.

Syntax: datatype arrayname[size]={list of values};

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

Organization of the elements in memory



The index begin with 0(zero) and ends with one less than the size of the array.

Runtime accessing:

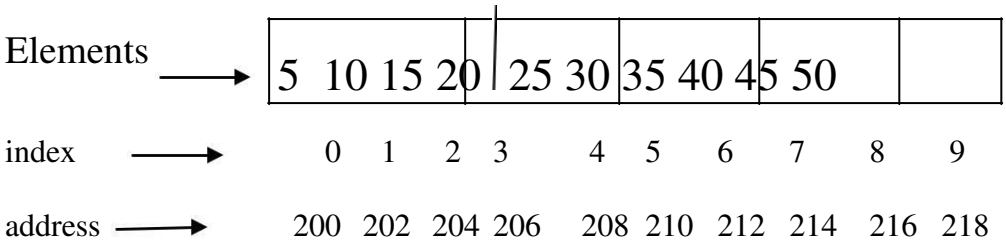
Read the array elements at time of execution from the keyboard using input statements.

Example:

```
int a[10];
for(i=0;i<=9;i++)
{
scanf("%d",&a[i]);
}
```

The above loop execute 10times and input the different array elements. For example entered elements are 5 10 15 20 25 30 35 40 45 50.

Organization of the elements in memory.



How to read and write the array elements.

```
#include<stdio.h>
main()
{
int a[10],i;
clrscr();
printf("Enter array elements \n");
for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
}
printf("Display the array elements \n");
for(i=0;i<10;i++)
{
printf("%d",a[i]);
}
getch();
}
```

Storing elements:

Declaration and definition only reserve space for the elements in the array. No values are stored. If we want to store values in the array, we must initialize the elements, read values from the keyboard or assign values to each individual element. int sample[5]={22,55,33,77,88};

Sample array have five space for store the five different elements. The elements are stored in array based on the index. For example 22 stored at sample[0], 55 stored at sample[1], 33 stored at sample[2], 77 stored at sample[3] and 88 stored at sample[4].

Types of arrays: There are three types‟ arrays.

1. One-dimensional array(1-D)
2. Two-dimensional array(2-D)
3. Multi dimensional array

One-dimensional array:

In one dimensional array, the organization of data is only one direction, because 1-D has only one dimension.

Example: datatype arrayname[size];

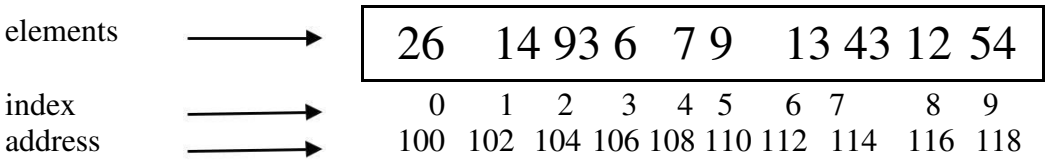
1-D integer array:

int a[10];

In the above statement array name „a‟ has 10 integer locations, „a‟ occupied memory is 20 bytes (2*10).

Organization of the elements in memory.

a[0] a[1] --------- a[9]



1-D character array(string):

A group of characters can be stored in a character array. 1-D character arrays are also called as strings

char str[20];

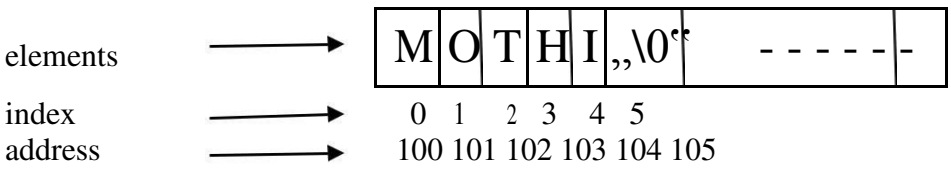
Above statement, arrayname str has 20 character elements. str occupied memory is 20 bytes(1*20).

Example1: char str[20]={„v‟,‟i‟,‟j,‟a‟,‟y‟,‟a‟};

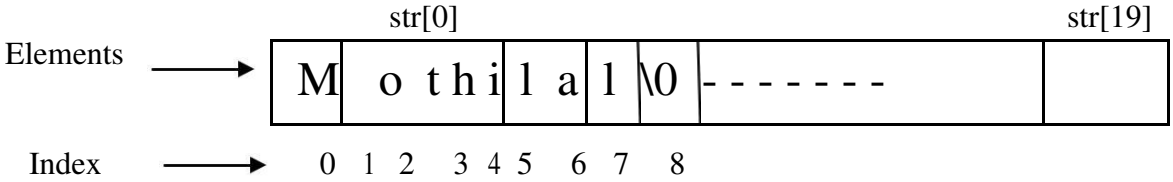
Example2: char str1[20]=”vijaya Prasad”;

Organization of elements in memory for str

str[0] str[19]



Organization of elements in memory for str1



Declaration and initialization of 1-D:

Declaration:

Syntax: datatype arrayname[size];

Example: int a[10];

Initialization:

Designtime: datatype arrayname[size]={list of values}; int a[10]={1,2,3,4,5,6,7,8,9,20};

Runtime:

```
int a[5],i;
printf("Enter array elements");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
```

```
}
```

Examples for 1-D:

Write a c-program to read and print array elements.

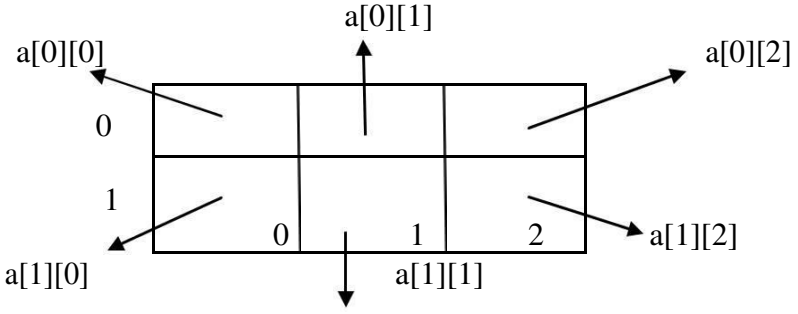
```
#include<stdio.h>
main()
{
int a[10],i;
clrscr();
printf("Enter array elements \n");
for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
}
printf("Entered array elements are\n");
for(i=0;i<10;i++)
{
printf("4%d",a[i]);
}
getch();
}
```

Write c-program to read and print your name.

```
main()
{
char name[20]="vijaya";
clrscr();
printf("My name is : %s", name);
getch();
}
```

Two dimensional array(2-D):

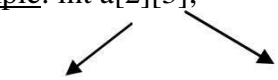
Many application require that data be stored in more than one dimension. One common example is a table. Which is an array that consists of rows and columns.
Syntax: datatype arrayname[row-size][column-size]; Example: int a[2][3];



Declaration of 2-D:

Syntax: datatype arrayname[row-size][column-size];

Example: int a[2][3];



rows columns

Have 12 bytes(6*2 integers) gets allocated to „a" initializing a 2-D array.

At design time:

```
datatype arrayname[r-size][c-size]={list values};

int a[2][3]={5,10,15,20,25,30};
            Or
int a[2][3]={{5,10,15},{20,25,30}};
```

At runtime:

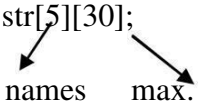
```
datatype arrayname[r-size][c-size];
int a[2][2],i,j;
for(i=o;i<2;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&a[i][j]);
}
}
```

2-D character arrays:

2-D chaarater arrays are nothing but an array(collection) of strings.

Declaration:

```
char str[5][30];
```

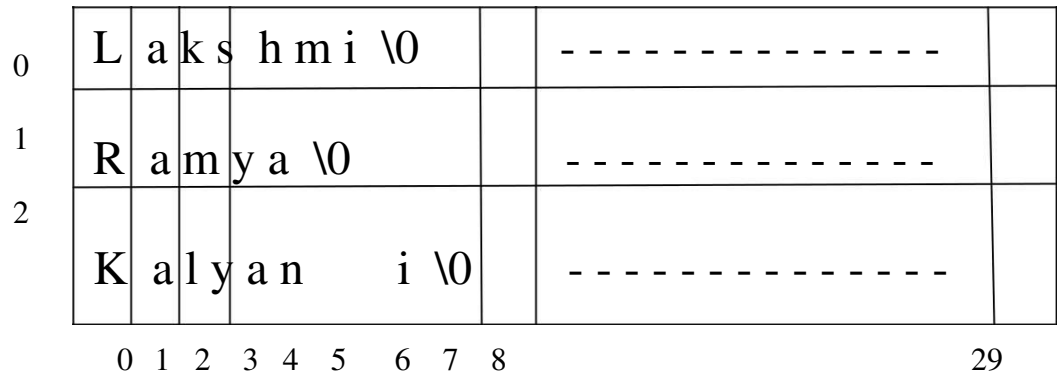


max. no.of names max. size of each name

Initialization:

```
char str[3][30]={"Lakshmi","Ramya","Kalyani"};
```

Stored on:



Examples on Two-dimensional array:

1. Read and print array elements using 2-D

```
main()
{
        int a[2][2],i,j;
        printf("Enter array elements
        \n"); for(i=0;i<2;i++)
        {
          for(j=0;j<2;j++)
          {
                scanf("%d",&a[i][j]);
          }
        }
        printf("Entered elements are \n");
        for(i=0;i<2;i++)
        {
          for(j=0;j<2;j++)
          {
            printf("4%d",a[i][j]);
          }
        }
getch();
}
```

→ Multidimensional arrays:

Multidimensional arrays can have three, four or more dimensions below diagram shows an array of three dimensions. Note the terminology used to describe the array. The first dimension is called a plane which consists of rows and columns. Arrays of four or more dimensions can be created and used but they are difficult to draw c language takes the three dimensional array to be an array of two dimensional arrays. It considers the two dimensional array to be an array of one dimensional arrays. In other words a three dimensional arrays in c is an array of arrays of arrays.

→ Declaring multidimensional arrays:

Declaration tells the compiler the name of the arrays, the type of each element and size of each dimension. The size of the fixed length array is a constant and must have a value at compilation time.

```
int table[planes][rows][cols];
```

→ Initialization:

Declaration and definition only reserve space for the elements in the array. No values will be stored in the array. If we want to store values, we must either initialize the elements, read values from the keyboard or assign values to each individual element.

```
int table[3][5][4]=
 {
   {{0,1,2,3},{10,11,12,13},{20,21,22,23},{30,31,32,33},{40,41,42,43}}
   {{100,101,102,103},{200,201,202,203},{300,301,302,303},
{400,401,402,403},{500,501,502,503} }
{{110,111,112,113},{210,211,212,213},{310,311,312,313},{410,411,412,413},
                        {510,511,512,513}}
                       };
```

1. Finding number of words in a sentence.

```
void main()
{
char str[128];
int i,count=0;
clrscr();
puts("Enter a sentence")
gets(str);
for(i=0;str[i]!=0;i++)
{
if(str[i]==" „)
count++;
}
printf("Number of words=%d",count+1);
}
```

2. sample

```
program main()
{
int a[10],i;
clrscr();
printf("Enter array elements
        \n"); for(i=0;i<2;i++)
{
scanf("%d",&a[i]);
}
printf("Elements\t Index \t Address \n");
for(i=0;i<2;i++)
{
printf("%d\t %d\t %d\t\n",a[i],I,&a[i]);
}
getch();
}
```

3.<u>Asending order:</u>

```
 #include<stdio.h>
main()
{
int i,j,k;
float n[10],t=0;
clrscr();
printf("Enter any 10 number:  \n");
for(i=0;i<10;i++)
{
scanf("%f",&n[i]);
```

```
}
for(i=0;i<10;i++)
{
for(j=i+1;j<10;j++)
{
if(n[i]>=n[j])
{
t=n[i];
n[i]=n[j];
n[j]=t;
}
}
}
printf("The asending order of the given number is : \n");
for(i=0;i<=9;i++)
printf("%7.5f",n[i]);
getch();
}
```

4. Write c-program to generate first 10 fibonacci numbers

```
#include<stdio.h>
#include<conio.h>
    main()
{
int i,fib[10];;
clrscr();
fib[0]=0;
fib[1]=1;
        for(i=2;i<10;i++)
{
    fib[i]=fib[i-1]+fib[i-2];
    }
    for(i=0;i<2;i++)
    {
    printf("%d\n",fib[i]);
    }
    getch();
    }
```

5. Read array elements and print them into reverse order.

```
#include<stdio.h>
    main()
    {
    int a[10],i;
    clrscr();
    printf("Enter array elements
    \n"); for(i=0;i<10;i++)
    {
```

```
scanf("%d",&a[i]);
}
printf("The elements in reverse order\n");
for(i=9;i>=0;i--)
{
printf("%d",a[i]);
}
getch();
}
```

6. Largest and smallest.

```
main()
{
int number[10],i,large=0,small=32767;
clrscr();
printf("Enter array elements \n");
for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
}
for(i=0;i<10;i++)
{
if(number[i]>large)
large=number[i];
}
for(i=0;I,=10;i++)
{
if(number[i]<small)
small=number[i];
}
printf("Largest value is %d",large);
printf("Smallest value is %d",small);
getch();
}
```

7.

```
#include<stdio.h>
#include< string.h>
    main()
    {
    char str[20];
    int i;
    clrscr();
    printf("Enter a string in uppercase");
    gets(str);
    for(i=0;str[i]!="\0")
    {
    if(str[i]>="A"&&str[i]<="2")
    str[i]=str[i]+32;
```

```
}
printf("The converted string is %s",str);
getch();
}
```

C – Strings:

A c-string is a variable length array of characters that is delimited by the null character.

Storing strings:

In c, a string is stored in an array of characters. It is terminated by the null characters („\O")

H E L L O \0

String delimiter:

Why do we need a null character at the end of a string?

Answer is that a string is not a data type but a data structure. This means that its implementation is logical, not physical.

Difference between the strings and character arrays

H E L L O \0 H E L L O

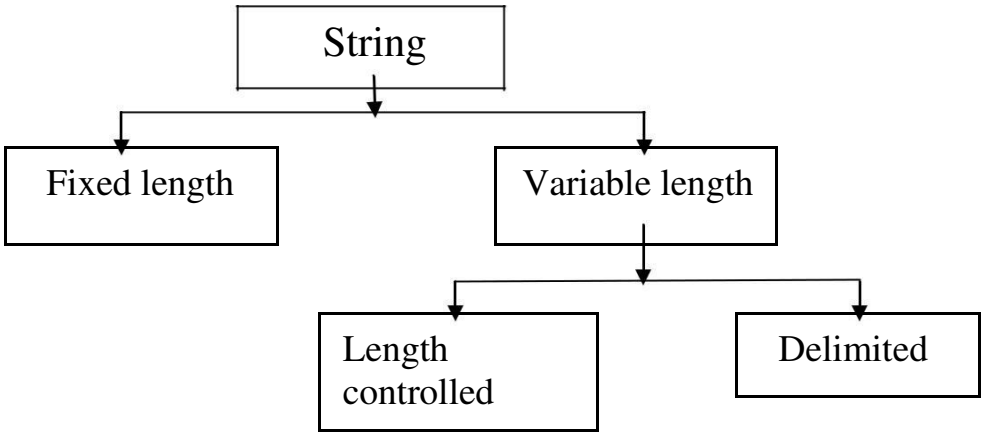
String literals: A string literal is a sequence of characters enclosed in double quotes.

Example:

"Have a nice day"
"Hello"
"Hai"
"Prasad"

Strings:

Strings concepts :



Fixed length strings: When implementing a fixed length string format the first decision is the size of the variable. If we make it too small, we can‟t store all the data. If we make it too big, we waste memory.

Example: 1. char str[15]=”Have a nice day”;
2. char str[100]=”Hai”;

Variable length strings: In variable length strings the user can use the required memory only. So reduce the memory wastage and data losses.

Example: char str[]=”good day”;

1. Length controlled strings: Length controlled strings add a count that specifies the number of characters in the string.
2. Delimited strings: Another techniques used to identify the end of the string is the delimiter at the ends of delimited strings.

Strings and characters: Data can be stored in two ways.
1. Data as character literal.
2. Data as string literal.

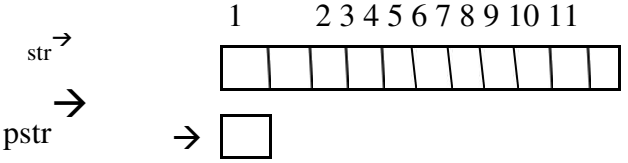
Data stored as character literal, we can use single quote marks.
Data stored as character literal, we can use double quote marks.

Example:

„a‟	a as character
“a”	a as string
” ”	empty character
“ “	empty string

Declare strings: String declaration defines memory for a string when it is declared as an array in local memory.

Syntax: 1. char str[9];
Syntax: 2. string pointer declaration.
char *pstr;



Initializing strings: We can initialize a string the same way that we initialize any storage structure by assigning a value to it when it is defined.

```
char str[9]=”good day”;
char name[ ]=”Prasad”;
```
String initialized as a character pointer.
```
char *pstr=”good day”;
```
String initialized as an array of characters.
```
char str[9]={„g‟,‟o‟,‟o‟,‟d‟,‟ „„‟d‟,‟a‟,‟y‟};
char str[10]={„n‟,‟I‟,‟c‟,‟e‟,‟
```
„„‟d‟,‟a‟,‟y‟,‟\O‟}; Strings are end with null character.

String input and output functions:

C provide two basic ways to read and write strings. First, we can read and write strings with the formatted input/output functions.

scanf/fscanf and printf/fprintf

Second, we can use a special set of string only functions.

getstring → (gets/fgets)

putstring → (puts/fputs)

Formatted string input:

scanf(): scanf is the general input statement. It is used to read character data or numeric data from the keyboard.

Syntax: scanf("format specifier", variablename);

Example: char str[9];
scanf("%s",str);

Formatted string output:

printf(): printf is the general output statement. It is used to write character data or numeric data on the output units.

Syntax: printf("Format specifier",variablename);

Example: char str[9];
str="Hai";
printf("%s",str);

Example program:

```
main()
{
char str[9];
clrscr();
printf("Enter your string");
scanf("%s",str);
printf("Your string is %s",str);
getch();
}
```

String functions (input and output):

String input: gets()

The function gets accepts the name of string as a parameter, and fills the string with characters that are input from the keyboard, till a newline character is encountered, at the end, the function gets appends a null terminator as must be done for any string and returns. The new line character is not added to the string.

Syntax: gets(variable);

String output: puts()

This function is used to print the string message on to the output screen and also display the stored string in variable.

Syntax: puts("string message");
puts(variable);

Example on gets() and puts():

```
#include<string.h>
```

```
main()
{
char str[10];
clrscr();
puts("Enter your name");
gets(str);
puts("Entered name is ");
puts(str);
getch();
}
```

Output: Enter your name
Prasad
Entered name is
Prasad.

Character input and output functions:

Character input functions:

1. getchar()
2. getche()
3. getch()

Above three statements used for read a single character from keyboard at a time.

getchar(): getchar() function is used read a character from the keyboard and assigned the character to variable, the variable stored the character.

Syntax: variable=getchar ();

Example program:

```
#include<stdio.h>
main()
{
char c;
printf("Enter your character\n");
c=getchar();
printf("Entered character is \n");
putchar(c);
getch();
}
```

getch():

It is also read the characters from the keyboard and passes it immediately to the program with echoing on the screen.

Example program:

```
#include<stdio.h>
main()
{
char ch;
clrscr();
printf("enter a character\n");
ch=getch();
printf("enter character is %c",ch);
getch();
```

```
}
```

getche():

Accepts a character and passes it to the program and echos(displays) the same character on the screen also.

Example program:

```
#include<stdio.h>
main()
{
char ch;
printf("Enter your character \n");
ch=getche();
printf("Entered character is \n");
putchar(ch);
getch();
}
```

Character output function:

putchar(): putchar() is the character output function it is used to print the stored character in the variabale on to the screen.

Syntax: putchar(variable);

Example program:

```
#include<stdio.h>
main()
{
char ch;
clrscr();
printf("Enter your character  \n");
ch=getchar();
printf("Your character is \n");
putchar(ch);
getch();
}
```

Standard C string library functions:

Function	Description
strlen()	Determines length of a string
strcpy()	Copy the string from one to another
strncpy()	Copies characters of a string to another string upto specified length
strcmp()	Compares two strings
stricmp()	Compare two strings(function does not discriminates between small and capital letter)
strlwr()	Converts uppercase letters to lowercase letters
strupr()	Converts lowercase letters to uppercase letters
strdup()	Duplicate a string

strchr()	First character of the given string
strrchr()	Last character of the given string
strstr()	Determines first occurrence of the given string in another string
strcat()	String concatenation, means second string appends to the first string.
strncat()	Appends source string to destination string up to specified length.
strrev()	String reverse, reverse the all character of a string
strset()	Sets all characters of string with a given argument or symbol
strnset()	Sets specified numbers of characters of string with a given number or symbol
strspn()	Finds up to what length two strings are identical
strpbrk()	Searches the first occurrence of a character in a given string and then it displays the string starting from that character.

strlen(): This functions returns the number of character of a string data as an integer.

Syntax: strlen(variable);

Example program:

```
#include< string.h>
main()
{
char str[20];
int i;
clrscr();
puts("Enter your string");
gets(str);
i=strlen(str);
printf("Length of the given string is %d",i);
getch();
}
```

strcpy(): This function is used to copy the data of one string to another string variable.

Syntax: strcpy(stringvariable,stringvariable1);

The string in the second position is the source string that contains the data to be copied. The string in the first position is destination string that receives the data.

Example program:

```
#include< string.h>
main()
{
char str[20],str1[20];
clrscr();
puts("Enter ur string");
gets(str);
strcpy(str1,str);
puts(str);
puts(str1);
getch();
}
```

strcat(): This function is used concatenate two strings.

Syntax: strcat(stringvariable,stringvariable1);

The string in the first position is called destination string and the string in the second position is called source string the source is joined with the destination and the combined string is stored the destination itself.

Example program:

```
#include<string.h>
main()
{
char str1[20],str2[20];
clrscr();
puts("Enter both string1 and string2");
gets(str1);
gets(str2);
strcat(str1,str2);
puts(str1);
puts(str2);
getch();
}
```

strcmp(): This function is used to compare two strings.

Syntax: strcmp(stringvariable1,stringvariable2);

The strcmp() functionreturn an integer value as the result.

1.result +ve : String variable1 is greater than string variable2.
2.result –ve : String1is lessthan string2.
3.result is zero : Both strings are equal.

Example program:

```
#include< string.h>
main()
{
int i;
char str1[20],str2[20];
clrscr();
puts("Enter both strings");
gets(str1);
gets(str2);
i=strcmp(str1,str2);
printf("Difference is %d",i);
getch();
}
```

strrev(): This function is used to get the reverse string of a given string.

Syntax: strrev(string);

Example program:

```
#include< string.h>
```

```
main()
{
char str[20];
clrscr();
puts("Enter a string");
gets(str);
strrev(str);
puts("Reverse string of the given string is");
puts(str);
getch();
}
```

strlwr(): This function is used to convert the uppercase string to lower case string.

Syntax: strlwr(string);

Example program:

```
#include< string.h>
main()
{
char str[20];
clrscr();
puts("Enter the string in uppercase");
gets(str);
strlwr(str);
puts(str);
getch();
}
```

strupr(): This function is used to convert the lowercase string to uppercase string.

Syntax: strupr(string);

Example program:

```
#include< string.h>
main()
{
char str[20];
clrscr();
puts("Enter the string in lowercase");
gets(str);
strlwr(str);
puts(str);
getch();
}
```

Sample programs on Strings:

1. To check the given string is palindrome or not.

```
#include< string.h>
main()
```

```
{
char str[20]str1[20];
clrscr();
puts("Enter a string ");
gets(str);
strcpy(str1,str);
strrev(str);
i=strcmp(str,str1);
if(i==0)
puts("Given string is palindrome");
else
puts("Given string is not palindrome");
getch();
}
```

2. Find a word in a string using string function.

```
#include< string.h>
main()
{
char str[20]str1[20];
clrscr();
Ppts("Enter a string ");
gets(str);
puts("Enter find string");
gets(str1);
if(strstr(str,str1))
puts("Yes it is in the string ");
else
puts("Not found");
getch();
}
```

UNIT-6

POINTER

A pointer provides a way of accessing a variable without referring to the variable directly. A pointer variable is a variable that holds the memory address of another variable. The pointer does not hold a value. A pointer points to that variable by holding a copy of its address. It has two parts

1) the pointer itself holds the address
2) the address points to a value

Uses of pointers:

1) Call by Address
2) Return more than one value from a function
3) Pass array and string more conveniently from one function to another
4) Manipulate arrays more easily
5) Create complex data structures
6) Communicate information about memory
7) Fast compile

Declaring Pointer: The pointer operator a variable in C is '*' called "value at address" operator. It returns the value stored at a particular memory the value at address operator is also called "indirection" operator a pointer variable is declared by preceding its name with an asterisk(*) .

Syntax: *datatype *pointer_variable;*

Where datatype is the type of data that the pointer is allowed to hold and pointer_variable is the name of pointer variable.

Example:

```
int *ptr;
char *ptr;
float *ptr;
```

Sample program:

```
main()
{
int *p;
float *q;
double *r;
printf("Size of the integer pointer %d",sizeof(p));
printf("size of the float pointer %d",sizeof(q));
printf("size of double pointer %d",sizeof(r));
}
```

Initializing Pointers:
A pointer must be initialized with a specified address operator to its use. For example to store the address of I in p, the unary & operator is to be used. P=&I;

Sample program:

```
main()
{
int num=5;
int *ptr=&num;
printf("the address num is %p",&num);
printf("the address num is %p",ptr);
}
```

Address Operator(&): It is used as a variable prefix and can be translated as address of this & variable can be read as "address of variable".

```
int *p;
int a;
p=&a;
```

Pointer arithmetic or Address Arthmatic:
Some of the valid operation on the pointer

1) Assignment of pointers to the same type of pointers.
2) Adding or subtracting a pointer and an integer
3) Subtracting or comparing two pointers
4) Incrementing or decrementing the pointers
5) Assigning the value 0 to the pointer variable and comparing 0 with pointer.

Invalid operation for pointer

1) adding of two pointers
2) multiplying a pointer with a number
3) dividing a pointer with a number

1) **Assignment:** pointers with the assignment operators can be used if the following conditions are
   1) lefthand side operand is a pointer is a pointer and right hand operand is a null pointer constant.
   2) Incompatible type of pointer assignment
   3) Both the operands are pointers to compatibles.

Sample program:

```
main()
{
int i;
int *ip;
void *vp;
ip=&i;
vp=ip;
ip=vp;
if(ip!=&i)
printf("Compiler error");
else
printf("no compiler error");
}
```

2) Addition or Subtraction with integers

Addition: the operator + performs the addition operation between the pointer and number. For example p is a pointer, pointer p adds with value s means (p+s).

Example

```
main()
{
int a[]={10,12,6,7,2};
int i;
int sum=0;
int *p;
p=a;
for(i=0;i<5;i++)
{
sum=sum+*p;
p++;
}
printf("%d",sum);
getch();
}
```

3) Subtraction: the operator '-' performs the subtraction operation between the pointer and number. For example p is a pointer. Pointer p subtract value 5 mean (p-5).

Example:

```
main()
{
double a[2],*p,*q;
p=a;
p=p-1; printf("%d",(int)q-
(int)p);
}
```

4) Comparing pointers: C allows pointer to be compared with each other. If two pointer compare equal to each other.

```
main()
{
int a,*p,*q; p=&a;
q=&a; if(*p==*q)
printf("both are equal");

}
```

5) Assign the value 0 to the pointer and compare 0 with pointer: P declare as a pointer naturally it stores the address of another variable. Suppose 0 assigned to pointer, like p=0; and also pointer variable p compared with 0.

For example,

```
main()
{
int *p;
p=0;
if(p==0)
printf("P contains zero");
else
printf("p contains a non-zero value");
}
```

Pointers to pointers: Pointers to pointers concept supports the pointer of pointer, means pointer stores address of another variable. Pointer of pointer stores address of another pointer.

```
int a=5;
int *p;
int **q;
p=&a;
q=*p;
```

Hera, a is an integer variable it has value 5. P is integer pointer it has address of a. q is integer pointer of pointer it has address of p.

```
main()
{
int a=5;
int *p,**q;
p=&a;
q=&p;
printf("*p=%d",*p);
printf("**q=%d",**q);
}
```

Pointers to functions: Pointers are pointer i.e., variable which point to the address of a function. A running program is allocated a certain space in the main memory.

Declaration of a pointer to a function:

Syntax: *returntype(*function_pointer_name)(argument1,argument2,---);*

In the following example, a function pointer named fp is declared. It points to the functions that take one float, and two char and return as int.

*int (*fp)(float,char,char);*

Initialization of Function Pointers:

It is quite easy to assign the address of a function to a function pointer. It is optional to use the address operator & in front of the function's name. for example if add() and sub() are declared as follows.

```
int add(int,int);
int sub(int,int);
```

The names of the functions add and sub is pointers to those functions. These can be assigned to pointer variables.

```
fpointer=add;
fpointer=sub;
```

Calling of a function using a function pointer: in c there are two ways of calling a function using a function pointer. Use the name of the function pointer instead of the name of the function pointer instead of the name of the function or explicitly deference it.

```
Result=fpointer(4,5);
Result2=fpointer(6,2);
```

Example:

```
int (*fpointer)(int,int);
int add(int,int);
int sub(int,int);
main()
{
fpointer=add;
printf("%d",fpointer(4,5));
fpointer=sub;
printf("%d",fpointer(6,2));
}
int add(int a,int b)
{
return (a+b);
}
int sub(int a,int b)
{
return (a-b);
}
```

Passing a function to another function:

A function pointer can be passed as a function's calling argument.

Example:

```
double sum(double f(double,double),int m,int n)
{
int k;
double x;
double s=0.0;
printf("Enter the value of x");
scanf("%lf",&x);
for(k=m;k<=n;++k)
s+=f(k,x);
return s;
}
```

The following is an equivalent header to the function.

```
double sum(double (*f)(double),int m,int n)
{
same as above
}
```

Arrays of Pointers: an array of pointers can be declared very easily.

```
int *p[10];
```

This declares an array of 10 pointers each of which points to an integer. The first pointer is called p[0], second is p[1] and so on up to p[9].

Example:

```
main()
{
int *p[10];
int a=5,b=6,c=10;
p[0]=&a;
p[1]=&b;
p[2]=&c;
printf("Address of a %p",p[0]);
printf("Address of b %p",p[1]);
printf("Address of c %p",p[2]);
}
```

Pointers to an Array: we can declare a pointer to a simple integer value and make it point to the array asa is done normally.

```
int v[5]={1004,2201,3000,432,500};
int *p=v;
printf("%d",*p);
```

This piece of code displays the number, which the pointer p points to, that is the first number in the array namely 1004.

P++ this instruction increases the pointer so that it points to the next element of the array.

*printf(%d",*p);* this statement prints value 2201.

Pointers and Multidimensional Arrays:

It is usually best to allocate an array of pointers, and then initialize each pointer to a dynamically allocated row. Here is an example:

```
#include<stdio.h>
#include<stdio.h>
#define row 5
#define col 5
main()
{
int **arr,i,j;
arr=(int**)malloc(row*sizeof(int*));
if(!arr)
{
printf("out of memory\n");
exit(EXIT_FAILURE);
}
for(i=0;i<row;i++)
{
arr[i]=(int*)malloc(sizeof(int)*col);
if(!arr[i])
{
printf("Out of memory\n");
exit(EXIT_FAILURE);
}
}
printf("\nEnter the elements of the matrix row by row\n");
for(i=0;i<row;i++)
for(j=0;j<col;++j)
scanf("%d",&arr[i][j]);
printf("the matrix is as follows…\n");
for(i=0;i<row;++j)
{
printf("\n");
for(j=0;j<col;++j)
printf("%d\t",arr[i][j]);
}
}
```

With exit(), status is provided for the calling process as the exit status of the process.

STATUS	INDICATES
EXIT_SUCCESS	Normal program terminals
EXIT_FAILURE	Abnormal program termination

Generic pointer (or) void pointer:

void pointer in c is known as generic pointer. Literal meaning of generic pointer is a pointer which can point type of data.
Example:

```
void *ptr;
```

Here ptr is generic pointer.

Important points about generic pointer in c?
We cannot dereference generic pointer.

```
#include<stdio.h>
#include <malloc.h>
int main(){
        void *ptr;
        printf("%d",*ptr);
return 0;
}
```

Output: Compiler error

2. We can find the size of generic pointer using sizeof operator.

```
#include <string.h>
#include<stdio.h>
int main(){
        void *ptr;
        printf("%d",sizeof(ptr));
return 0;
}
```

Output: 2
Explanation: Size of any type of near pointer in c is two byte.

3. Generic pointer can hold any type of pointers like char pointer, struct pointer, array of pointer etc without any typecasting.

Example:

```
#include<stdio.h>
int main(){
        char c='A';
        int i=4; void
        *p; char
        *q=&c; int
        *r=&i; p=q;

        printf("%c",*(char *)p);
        p=r; printf("%d",*(int
        *)p);
return 0;
}
```

Output: A4

4. Any type of pointer can hold generic pointer without any typecasting.

5. Generic pointers are used when we want to return such pointer which is applicable to all types of pointers. For example return type of malloc function is generic pointer because it can dynamically allocate the memory space to stores integer, float, structure etc. hence we type cast its return type to appropriate pointer type.

Examples:

1.
```
char *c;
c= (char *)malloc(sizeof(char));
```

2.
```
double *d;
d= (double *)malloc(sizeof(double));
```

3.
```
Struct student{
        char *name;
        int roll;
};
Struct student *stu;
Stu=(struct student *)malloc(sizeof(struct student));
```

NULL pointer in c programming

NULL pointer:

Literal meaning of NULL pointer is a pointer which is pointing to nothing. NULL pointer points the base address of segment.

Examples of NULL pointer:

```
1. int *ptr=(char *)0;
2. float *ptr=(float *)0;
3. char *ptr=(char *)0;
4. double *ptr=(double *)0;
5. char *ptr='\0';
6. int *ptr=NULL;
```

What is meaning of NULL?

NULL is macro constant which has been defined in the heard file stdio.h, alloc.h, mem.h, stddef.h and stdlib.h as

```
#define NULL 0
```

Examples:What will be output of following c program?

```
#include <stdio.h>
int main(){
        if(!NULL)
        printf("I know preprocessor");
        else
        printf("I don't know preprocessor");
 return 0;
}
```

Output: I know preprocessor

Explanation:

!NULL = !0 = 1

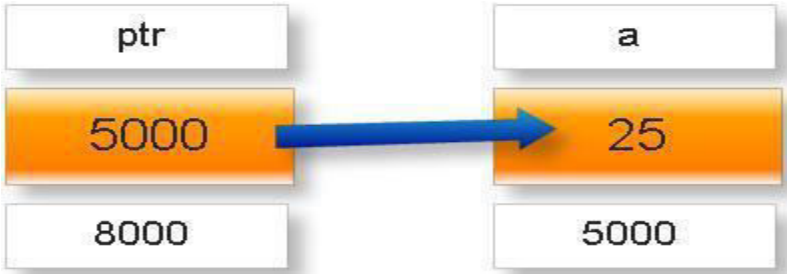
In if condition any non zero number mean true.

Dangling pointer problem in c programming

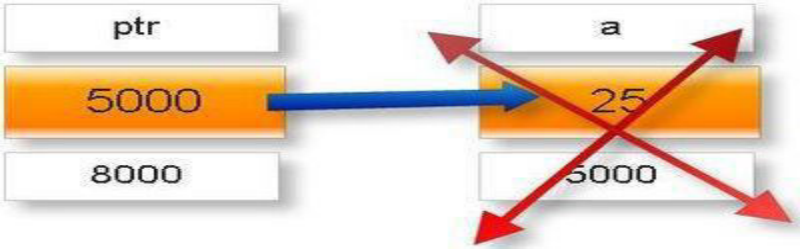
1. Dangling pointer:

If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem.

Initially:



Later:



For example:

(q)What will be output of following c program?

```
#include<stdio.h>
int *call();
void main(){

        int *ptr;
        ptr=call();
        fflush(stdin);
        printf("%d",*ptr);
}
int * call(){
        int x=25;
        ++x;
        return &x;
}
```

Output: Garbage value

Note: In some compiler you may get warning message **returning address of local variable or temporary**

Explanation: variable x is local variable. Its scope and lifetime is within the function call hence after returning address of x variable x became dead and pointer is still pointing ptr is still pointing to that location.

Solution of this problem: Make the variable x is as static variable.
In other word we can say a pointer whose pointing object has been deleted is called dangling pointer.

```
#include<stdio.h>
int *call();
void main(){
        int *ptr;
        ptr=call();

        fflush(stdin);
        printf("%d",*ptr);
}
int * call(){
        static int x=25;
        ++x;
        return &x;
}
```

Output: 26

Character Pointers and Functions

Since text strings are represented in C by arrays of characters, and since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C.

C does not provide any operators for processing an entire string of characters as a unit.We've said this sort of thing before, and it's a general statement which is true of all arrays. Make sure you understand that in the lines

```
                char *pmessage;
        pmessage = "now is the time";
        pmessage = "hello, world";
```

all we're doing is assigning two pointers, not copying two entire strings.

We also need to understand the two different ways that string literals like "now is the time" are used in C. In the definition

```
        char amessage[] = "now is the time";
```

The string literal is used as the initializer for the array amessage. amessage is here an array of 16 characters, which we may later overwrite with other characters if we wish. The string literal merely sets the initial contents of the array. In the definition

```
char *pmessage = "now is the time";
```

on the other hand, the string literal is used to create a little block of characters somewhere in memory which the pointer pmessage is initialized to point to. We may reassign pmessage to point somewhere else, but as long as it points to the string literal, we can't modify the characters it points to.

As an example of what we can and can't do, given the lines

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

we could say

```
amessage[0] = 'N';
```

to make amessage say "Now is the time". But if we tried to do

```
pmessage[0] = 'N';
```

The first function is strcpy(s,t), which copies the string t to the string s. It would be nice just to say s=t but this copies the pointer, not the characters.

This is a restatement of what we said above, and a reminder of why we'll need a function, strcpy, to copy whole strings.

Once again, these code fragments are being written in a rather compressed way. To make it easier to see what's going on, here are alternate versions of strcpy, which don't bury the assignment in the loop test. First we'll use array notation:

```
void strcpy(char s[], char t[])
{
        int i;
        for(i = 0; t[i] != '\0'; i++)
                s[i] = t[i];
        s[i] = '\0';
}
```

Note that we have to manually append the '\0' to s after the loop. Note that in doing so we depend upon i retaining its final value after the loop, but this is guaranteed in C, as we learned in Chapter 3.

Here is a similar function, using pointer notation:

```
void strcpy(char *s, char *t)
{
        while(*t != '\0')
        *s++ = *t++;
        *s = '\0';
}
```

Again, we have to manually append the '\0'. Yet another option might be to use a do/while loop.

Dynamic memory allocation

Dynamic memory allocation is the practice of assigning memory locations to variables during execution of the program by explicit request of the programmer. dynamic allocation is a unique feature to C

Finally, the dynamically allocated memory area, the area is secured in a location different from the usual definition of a variable. Used in the dynamic memory allocation area is called the heap..

The following functions are used in C for purpose of memory management.

1.malloc() 2.calloc() 3.realloc() 4.free()

malloc()

The **malloc**() function dynamically allocates memory when required. This function allocates 'size' byte of memory and returns a pointer to the first byte or NULL if there is some kind of error.

Format is as follows.

void * malloc (size_t size);

The return type is of type void *, also receive the address of any type. The fact is used as follows.

double * p = (double *) malloc (sizeof (double));

The size of the area using the sizeof operator like this. The return value is of type void *, variable in the receiving side can be the pointer of any type in the host language C .In C, a pointer type to void * type from another, so that the cast automatically, there is no need to explicitly cast originally.

With this feature, you get a pointer to an allocated block of memory. Its structure is:

code:

pointer = (type) malloc (size in bytes);

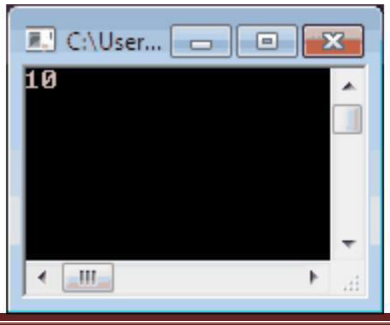
An example:

code:

```
int * p;
p = (int *) malloc (sizeof (int));
* p = 5;
```

The following example illustrates the use of malloc() function.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int a,*ptr;
a=10;
```



```
ptr=(int*)malloc(a*sizeof(int));
ptr=a;
printf("%d",ptr);
free(ptr);
getch();
}
```

calloc() function

The calloc function is used to allocate storage to a variable while the program is running. This library function is invoked by writing calloc(num,size).This function takes two arguments that specify the number of elements to be reserved, and the size of each element in bytes and it allocates memory block equivalent to num * size . The important difference between malloc and calloc function is that calloc initializes all bytes in the allocation block to zero and the allocated memory may/may not be contiguous. For example, an int array of 10 elements can be allocated as follows.

```
int * array = (int *) calloc (10, sizeof (int));
```

Note that this function can also malloc, written as follows.

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

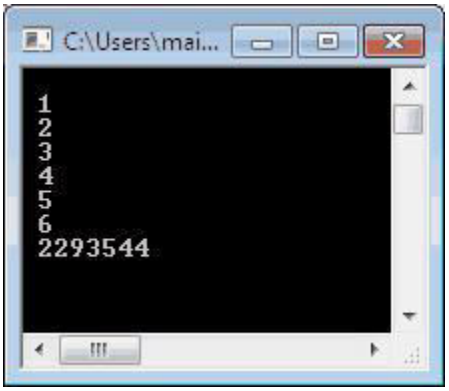
```
ptr = malloc(10 * sizeof(int));
```

is just like this:

```
ptr = calloc(10, sizeof(int));
```

The following example illustrates the use of

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int *ptr,a[6]={1,2,3,4,5,6};
int i;
ptr=(int*)calloc(a[6]*sizeof(int),2);
for(i=0;i<7;i++)
{
printf("\n %d",*ptr+a[i]);
}
free(ptr);
getch();
}
```



realloc()

With the function realloc, you can change the size of the allocated area once. Has the following form.

void * realloc (void * ptr, size_t size);

The first argument specifies the address of an area that is currently allocated to the size in bytes of the modified second argument. Change the size, the return value is returned in re-allocated address space. Otherwise it returns NULL.

The address of the source address changed, but the same could possibly be different, even if the different areas of the old style, because it is automatically released in the function realloc, for the older areas it is not necessary to call the free function. #include<alloc.h>

```
#include<string.h>
main()
{
char *str;
clrscr();
str=(char*)malloc(6);
str=("india");
printf("str=%s",str);
str=(char*)realloc(str,10);
 strcpy(str,"hindustan");
printf("\nnow str=%s",str);
free(str);
getch();
}
```

free() function

Next, how to release the reserved area, which will use the function free. Has also been declared in stdlib.h, which has the following form. void free (void * ptr);

The argument specifies the address of a dynamically allocated area. You can then free up the space. Specify an address outside the area should not be dynamically allocated. Also, if you specify a NULL, the free function is guaranteed to do nothing at all

So, an example program that uses functions malloc and free functions in practice.

```
# include<stdlib.h>
# include<conio.h>
int main (void)
{
int * p;
p = (int *) malloc (sizeof (int)); /* partitioning of type int */
if (p == NULL) * / failed to reserve area * /
{
printf ("Failed to allocate space for% d bytes", sizeof (int));
return 1;
}
* P = 150;
printf ("% d \ n", * p);
free (p); * / free area * /
return 0;
}
```

COMMAND LINE ARGUMENTS

It is possible to pass arguments to C programs when they are executed. The brackets which follow main are used for this purpose. *argc* refers to the number of arguments passed, and *argv[]* is a pointer array which points to each argument which is passed to main. A simple example follows, which checks to see if a single argument is supplied on the command line when the program is invoked.

```
#include <stdio.h>

main( int argc, char *argv[] )
{
        if( argc == 2 )
                printf("The argument supplied is %s\n", argv[1]);
        else if( argc > 2 )
                printf("Too many arguments supplied.\n");
        else
                printf("One argument expected.\n");
}
```

Note that **argv[0]* is the name of the program invoked, which means that **argv[1]* is a pointer to the first argument supplied, and **argv[n]* is the last argument. If no arguments are supplied, *argc* will be one. Thus for n arguments, *argc* will be equal to n + 1. The program is called by the command line,

myprog argument1

Example : commandline.c

```
#include<stdio.h>
main(int argc,int *argv[])
{
int i;
printf("No of arguments %d",argc);
for(i=0;i<argc;i++)
{
printf("\n %s",argv[i]);
}
}
```

```
C:\CSE>filename arg1 arg2 …….. argn
C:\CSE>commandline hello how are u
No of arguments 5
commandline
hello
how
are
u
```

DERIVED DATATYPES:

C provides facilities to construct user defined datatypes. A user defined datatypes may also be called a derived datatype. Some of the datatypes are…

1. structures
2. unions
3. typedef
4. arrays
5. enumerators…etc.,

STRUCTURE

→ Def. of structure:

Structure is the user defined datatype that can hold a heterogeneous datatypes. Structures are constructing with "struct" keyword and end with semicolon…

→ Declaration of structures and structure variables:

Structures are declared by using struct keyword followed by structure name followed by body of the structure. The variables or members of the structure are declared with in the body.

Syntax:

```
struct structname
  {
   datatype member1;
    .
     .
     .
   datatype member n;
   };
   struct structurename <struct variable1>….<struct variable n>;
```

In the above syntax structurename is the name of the structure. The structure variables are the list of the variable names separated by comas. Variables declared with in the braces called members or variables.

Example:

```
struct country
 {
   char name[30];
    int population;
    char language[15];
  } India,japan,England;
```

In the above example country is the structure name and structure members are name, population, language and structure variable are India, Japan and England.

→ Initialisation of structure:

This consists of assigning some constants to the constants to the members of the structure. Default values for integer and float datatype members are zero. For char member default value '\0' (null).

Syntax:

```
struct struct name
{
datatype member1;
.
.
.
datatype member n;
}<structvariable>={constant1,constant2…}
```

Example:

```
#include<stdio.h>
struct item
{
        int count;
        float avgweight;
        int date,month,year;
} batch={2000,25.3,07,11,2010};
main()
{
printf("\n count=%d",batch.count);
printf("\n averageweight=%f",batch.avgweight);
printf("\nmfg-date=%d%d%d",batch.date,batch.month,batch.year);
}
```

→ Accessing the member of a structure:

The members of the structure can be accessed by using "." Operator(dot operator).

Syntax:

<structurevariable>.<membername>;

The .(dot) operator selects a particular member from a structure.

Example:

```
struct list
    {
      int a;
      float b;
    } s,s1;
```

Accesing the a,b from the structure list is

s.a, s.b scanf("%d%f",&s.a,&s.b);

s1.a, s1.b scanf("%d%f",&s1.a,&s1.b);

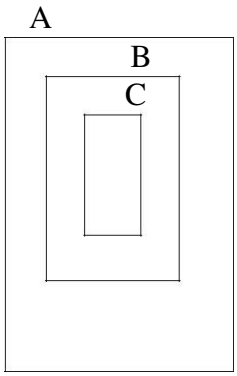
same as printing values

printf("%d%f",s.a,s.b);

printf("%d%f",s1.a,s1.b);

→ Nested structures:

A structure can be placed with in another structure. In other words structures can contain other structures as members. A structure with in a structure means nested structures.



Outermost structure (named A)
Inner structure (named B)
Innermost structure (named C)

Example:

```
#include<stdio.h>
struct b
{
int x;
struct b
{
int y;
struct c
{
int  z
}c1;
}b1;
}a1;
main()
{
        a1.x=10;
        a1.b1.y=20;
        a1.b1.c1.z=30;
```

```
        printf("x value %d",a1.x);
        printf("y value is %d",a1.b1.y);
        printf("z value is %d:,a1.b1.c1.z);
        getch();
}
```

→ Arrays of structures:

This means that the structure variable would be an array of objects. Each of which contains the number elements declared within the structure construct.

Syntax:

```
struct <structname>
{
            <datatype member1>;
<datatype member2>;
.
.
.
<datatype member n>;
}<structure variable[index];
```

(or)

```
struct <structname><structurevariable>[index];
```

Here the term index specifies the number of array objects.

Example:

```
#include<stdio.h>
struct Prasad
{
int a;
float b;char c;
};
main()
{
struct Prasad p[3];
int i;
clrscr();
printf("enter details\n");
for(i=0;i<3;i++)
scanf("%d%f%c",&p[i].a,&p[i].b,&p[i].c);
printf("entered details are \n");
for(i=0;i<3;i++)
printf("%d%f%c",p[i].a,p[i].b,p[i].c);
getch();
}
```

→ Initialising arrays of structures:

Example:

```
struct student
{
char name[30];
int rno,age;
};
main()
{
int i;
struct student
s[3]={{“prassad”,1201,30},{“raju”,530”,29},{“rani”,430,30}};
printf(“details fro students”);
for(i=0;i<3;i++)
printf(“\n %s\t %d\t%d”,s[i].name,s[i].rno,s[i].age);
getch();
}
```

→ Size of the structure:

Size of the structure specifies the structure occupied how many bytes using “sizeof” keyword..

Example:

```
struct student
{
char name[20];
int a[10];
float b[20];
}s;
main()
{
clrscr();
printf(“the size of the structure is %d”,sizeof(s));
getch();
}
```

→ Structures and pointers:

A pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure this pointer variable takes four bytes of memory.

Syntax:

```
struct <structname>
{
<datatype member1>;
<datatype member2>;
.
.
<datatype membern>;
}*ptr;
```

Or

```
struct <structname> *ptr;
```

The pointer *ptr can be assigned to any other pointer of the same type and can be used to access the members of its structure using pointer ,access the structure members in two ways…

One is…

(*ptr).member1;

The bracket is needed to avoid confusion about '*' and '.' Operators.

Second is…

ptr→member1;

This is less confusing and better way to access a member in a structure through its pointer. The → operator, an arrow made out of a minus sign and a greater than symbol.

Example:

```
#include<stdio.h>
#include<conio.h>
struct abc
{
int a;
float b;
}*p;
main()
{
printf("initialization for  members/n");
p→a=10;
p→b=20.12;
printf("a=%d,b=%f",p→a,p→b);
printf("read the values in");
scanf("%d%f",&p→a,p→b);
printf("\n a=%d,b=%f",p→a,p→b);
}
```

→ Structures and functions:

An entire structure can be passed on a function arguments just like any other variables.when a structure is passed as an argument,each member of the structure is copied.

Syntax:

struct<structurename><functioname>(struct structrename structurevariable);

Example:

```
#include<stdio.h>
struct A
{
char ch;
int i;
float f;
};
void show(struct A);
main()
{
struct A x;
printf("in entre ch,i and f");
scanf("%c%d%f",&x.ch,&x.i,&x.f);
show(x);
}
void show(A b)
{
printf("%c%d%f",b.ch,b.i,b.f);
}
```

→ typedef:

The typedef keyword allows the programmer to create a new datatype name for an existing datatype. No new datatype is produced but an alternative name is given to a known datatype

Syntax:

typedef <existing datatype><new datatype,…>;

typedef statement does not occupy storage, it simply defines a new type.

```
typedef int id;
typedef float wt;
typedef char lw;
```

id is the new datatype name given to datatype int, while wt is the new datatype name given to the datatype float,a nd lw is the new datatype name given to datatype char…

```
id x,y,z;
wt p,m;
lw a,b;
```

x,y and z are variable names that are declared to hold int datatype.

Example:

```
main()
{
typedef int Prasad;
Prasad a,b,c;
a=10;b=20;
c=a+b;
printf("%d",c);
}
```

In structures

```
#include<stdio.h>
typedef struct point
{
int x;
int y;
}data;
main()
{
data lt,rt;
printf("enter x and y coordinates of left and right");
scanf("%d%d%d%d",&lt.x,&lt.y,&rt.x,&rt.y);
printf("left:x=%d,y=%d.right:x=%d,y=%d",lt.x,lt.y,rt.x,rt.y);
getch();
}
```

→ Union:

Union is the user defined datatype that can hold heterogeneous datatypes. Unions are construct with "union" keyword. Unions are end with semicolon.

Syntax:

```
union unionname
{
Member1;
Member2;
.
.
.
Member n;
}variable1,variable2…..variable n;
```

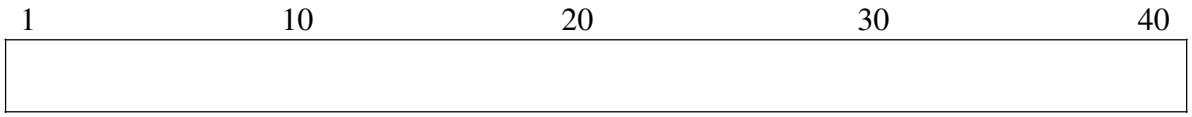
Union also has a union name,members and variable names. The union variables are declared with following syntax.

Union unionname variable1,variable2….variable n;

Example:

```
union mixed
  {
char ch[10];
 float marks[10];
  int no[10];
  }all;
```

Union members share the same storage



-------char----------
10bytes
----------int 20 bytes----------

----------float 40 bytes----------

→ Accessing and initializing members of union:

Syntax:

```
union unionname
{
 Member1;
  Member2;
    .
    .
    .
  Member n;
  }variable1,variable2….variable n;
```

For accessing the member of union using the following syntax.
unionvariable.member of union;
For initialization of union member is
unionvariable.member=constant;

Example:

```
#include<stdio.h>
#include<conio.h>
union exp
{
int i;
char c;
}var;
main()
{
```

```
var.i=65;
var.c=var.i;
printf("\n var.i=%d",var.i);
printf("\n var.c=%c",var.c);
getch();
}
```

→ Enumeration types:

Enumeration datatypes are data items whose values may be any number of a symbolically declared set of values the symbolically declared members are integer constants. The keyword enum is used to declare an enumeration type. Simply "enumerators are named integers"

The general form of the enumerator is

```
enum <enumerator name>
   {
    Member1;
     Member2;
       .
       .
       .
      Member n;
  }var1,var2,…var n;
```

The enum enumerator name specifies the userdefined type the member are integer constants, by default ,the first member, that is member1 is given the value0.

Example:

```
#include<stdio.h>
enum days
{
Mon,tue,wed,thur,fri,sat,sun
};
main()
{
enum days start,end;
start=tue;
end=sat;
printf("start=%d,end=%d",start,end);
start=64;
printf("\n start now is equal to %d",start);
getch();
}
```

Example:

```
#include<stdio.h>
enum rank
{
Ramu,ramesh,Prasad=10,vinaya,rani
}v;
main()
{
v=ramesh;
printf("ramesh rank is %d\n",r);
v=Prasad;
printf("prasad rank is %d\n",r);
v=rani;
printf("\n rani rank is %d",r);
getch();
}
```

→ Self referential structure:

A self referential structure is one which contains a pointer to it own type. This type of structure is also known as linked list. For example..

```
struct node
{
 int data;
 struct node *nextptr;
 };
```

Defines a datatype, struct node. A structured of type struct node has two members-integer member data and pointer member nextptr. Member nextptr points to a structure of type struct node-a structure of the same type as one being declared here, hence the term "self referential structure" member nextptr is refered to as link i.e nextptr can be used to "tie" a structure of type struct node to another structure of the sametype.

Example:

```
#include<stdio.h>
main()
{
struct stinfo
{
char name[20];
int age;
char city[20];
struct stinfo *next;
};
struct stinfo s1={"rani",25,"kkd"};
struct stinfo s2={"raju",27,"rjy"};
struct stinfo s3={"sita",29,"amp"};
s1.next=&s2;
```

```
s2.next=&s3;
s3.next=NULL;
printf("student name=%s,age=%d,city=%s",s1.name,s2.age,s3.city);
printf("student1 stored at %x \n",s1.next);
printf("student2 name=%s,age=%d,city=%s",s2.name,s2.age,s2.city);
printf("student2 stored at %x\n",s2.next);
printf("student3 name=%s,age=%d,city=%s",s3.name,s3.age,s3.city);
printf("student3 stored at %x\n",s3.next);
}
```

The next three statement in the program

```
s1.next=&s2;
s2.next=&s3;
s3.next=NULL;
```

Setup the linked list , with the next member of s1 pointing to s2 and the next member of s2 pointing to s3 and next member of s3 pointing to NULL.

→ Bitfields:

There are two ways to manipulate bits in c. one of the ways consists of using bitwise operators . the other way consists of using bit fields in which the definition and the access method are based on structure.

```
struct bitfield_tag
{
  unsigned int member1:bit_width1;
  unsigned int member2:bit_width2;
    .
    .
    .
  unsigned int member n:bit_width n;
};
```

Each bitfield for example 'unsigned int member1:bit_width1' is an integer that has a specified bit width. By this technique the exact number of bits required by the bit field is specified.

Exaample:

```
struct test
    {
      unsigned tx:2;
      unsigned rx:2;
      unsigned chk_sum:3;
      unsigned p:1;
    } status_byte;
```

Structure variable name status_byte containing four unsigned bitfields. The value assigned to a field must not be greater than its maximum storage value.

The assignment of the structure member is

```
Chk_sum=6;
```

Sets the bits in the field chk_sum on 110.

→ Difference between structure and union:

Structure	Union
1. structure is the user defined datatype . 2. structure are constructed with struct keyword. 3. structures members are stored in individual storage. 4. occupied more memory. 5. member accessing is difficult.	1. union is the user defined datatype. 2. union are construct with union keyword. 3. members share the same memory. 4. occupied less memory. 5. member accessing is easy.

→ Difference between array and pointers:

Array	Pointer
1. Array allocates space automatically. 2. It cannot be resized. 3. It cannot be reassigned. 4. sizeof(arrayname) gives the number of bytes occupied by array.	1. It is explicitly assigned to point to an allocated space. 2. It can be resized using realloc(). 3. It can be reassigned. 4. sizeof(p) returns the number of bytes used to store the point variable.

→ Difference between array and structure:

Array	Structure
1. Group of homogeneous elements. 2. Array is derived datatype. 3. An array behavior like a built in datatype. 4. Elements differ with index. 5. sizeof(arrayname) gives the number of bytes occupied by the array.	1. Group of heterogeneous elements. 2. Structure is a user defined datatype. 3. Structure behaves like a built in different datatypes. 4. members differ with datatype and member name. 5. sizeof(structurevariable) returns the number of bytes used to store the structure variable.

File management in C

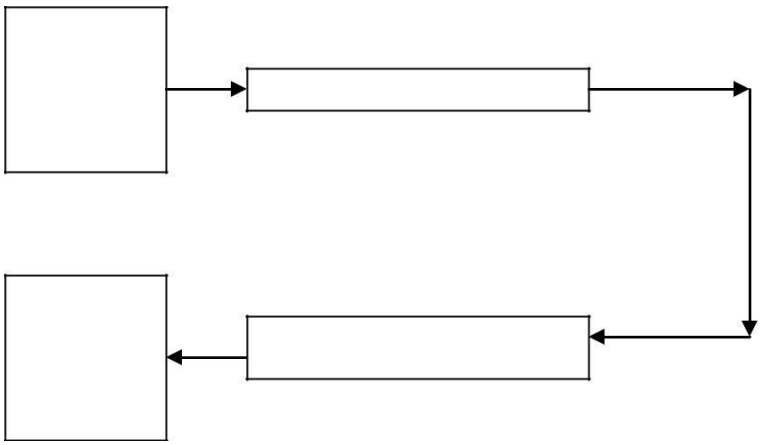
FILE:

A **file** is an external collection of related data treated as a unit. The primary purpose of a file is to keep a record of data. files are stored in auxiliary or secondary storage devices .

Buffer is a temporary storage area that holds data while they are being transferred to or from memory.

Streams

A stream can be associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.



Text and Binary Streams

C uses two types of streams: text and binary. A **text stream** consists of characters divided into lines with each line terminated by a newline(\n).A **binary stream** consists of data values such as integer, real or complex.

Creating a Stream

We create a stream when we declare it. The declaration uses the FILE type as shown in the following example the FILE type is a structure that contains the information needed for reading and writing a file.

```
FILE *spData;
```

In this example ,spData is a pointer to the stream.

Opening a File

When the file is opened , the stream and the file are associated with each other , and the FILE type is filled with the pertinent file information. the open function returns the address of the file type ,which is stored in the stream pointer variable, spData .

Using the Stream Name

After we create the stream ,we can use the stream pointer in all functions that need to access the corresponding file for input and output.

Closing the Stream

When the File processing is complete, we close the file. Closing the association between the stream name and file name . After the close, the stream is no longer available and any attempt to use it results in an error

The File Modes

Your program must open a file before it can access it. This is done using the fopen function, which returns the required file pointer. If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows
if ((output_file = fopen("output_file", "w")) == NULL) fprintf(stderr, "Cannot open %s\n", "output_file");
fopen takes two arguments, both are strings, the first is the name of the file to be opened, the second is an access character, which is usually one of r, a or w etc. Files may be opened in a number of modes, as shown in the following table.

File Modes	
r	Open a text file for reading.
w	Create a text file for writing. If the file exists, it is overwritten.
a	Open a text file in append mode. Text is added to the end of the file.
rb	Open a binary file for reading.
wb	Create a binary file for writing. If the file exists, it is overwritten.
ab	ary file in append mode. Data is added to the end of the file.
r+	Open a text file for reading and writing.
w+	Create a text file for reading and writing. If the file exists, it is overwritten.
a+	Open a text file for reading and writing at the end.
r+b or rb+	Open binary file for reading and writing.
w+b orwb+	Create a binary file for reading and writing. If the file exists, it is overwritten.
a+b or ab+	Open a text file for reading and writing at the end.

The update modes are used with fseek, fsetpos and rewind functions. The fopen function returns a file pointer, or NULL if an error occurs.
The following example opens a file, tarun.txt in read-only mode. It is good programming practice to test the file exists.

```
if ((in = fopen("tarun.txt", "r")) == NULL)
{
puts("Unable to open the file");
return 0;
}
```

File operation functions in C

File operation functions in C, Defining and opening a file, Closing a file, The getw and putw functions, The fprintf & fscanf functions

C supports a number of functions that have the ability to perform basic file operations, which include:

1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

- Real life situations involve large volume of data and in such cases, the console oriented I/O operations create two major problems
- It becomes burden and time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or computer is turned off.

File operation functions in C:

Function Name	Operation
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the begining of the file

Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the fielname, data structure, purpose.

The general format of the function used for opening a file is

```
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening the file. The mode does this job.

R open the file for read only.
W open the file for writing only.
A open the file for appending data to it.

Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

Closing a file:
The input output library supports the function to close a file; it is in the following format.
fclose(file_pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer. Observe the following program.

```
….
FILE *p1 *p2;
p1=fopen ("Input","w");
p2=fopen ("Output","r");
….
…
fclose(p1);
fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode.

```
c=getc(fp2).
```

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include< stdio.h >
main()
{
file *f1;
printf("Data input output");
f1=fopen("Input","w"); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf("nData outputn");
f1=fopen("INPUT","r"); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
}
```

Reading & Writing Files

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);
/*Example program for using getw and putw functions*/
#include< stdio.h >
main()
{
FILE *f1,*f2,*f3;
int number I;
printf("Contents of the data filenn");
f1=fopen("DATA","W");
for(I=1;I< 30;I++)
{
scanf("%d",&number);
if(number==-1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)/* Read from data file*/
```

```
{
if(number%2==0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("nnContents of the odd filenn");
while(number=getw(f2))!=EOF)
printf("%d%d",number);
printf("nnContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
fclose(f2);
fclose(f3);
}
```

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of fprintf is

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

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

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

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

The general format of fscanf is

```
fscanf(fp,"controlstring",list);
```

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

Example:

```
#include <stdio.h>
int main()
{
 FILE *fp;
 file = fopen("file.txt","w");
fprintf(fp,"%s","This is just an example :)");
fclose(fp);
 return 0;
}
```

Variables

Variables defined in the stdio.h header include:

Name	Notes
stdin	a pointer to a FILE which refers to the standard input stream, usually a keyboard.
stdout	a pointer to a FILE which refers to the standard output stream, usually a display terminal.
stderr	a pointer to a FILE which refers to the standard error stream, often a display terminal.

fflush

Defined in header <stdio.h>

int fflush(FILE *stream);

Causes the output file stream to be synchronized with the actual contents of the file. If the given stream is of the input type, then the behavior of the function is undefined.

Parameters

stream	the file stream to synchronize

Return value

Returns zero on success. Otherwise EOF is returned and the error indicator of the file stream is set.

Text Files in C

A file is for storing permanent data. C provides file operations in stdio.h. A file is viewed as a stream of characters. Files must be opened before being accessed, and characters can be read one at a time, in order, from the file.

There is a current position in the file's character stream. The current position starts out at the first character in the file, and moves one character over as a result of a character read (or write) to the file; to read the 10th character you need to first read the first 9 characters (or you need to explicitly move the current position in the file to the 10th character).

There are special hidden chars (just like there are in the stdin input stream), '\n', '\t', etc. In a file there is another special hidden char, EOF, marking the end of the file.

Using text files in C

1 DECLARE a FILE * variable

```
FILE *infile;
FILE *outfile;
```

2 OPEN the file: associate the variable with an actual file using fopen you can open a file in read, "r", write, "w", or append, "a" mode

```
infile = fopen("input.txt", "r");
if (infile == NULL) {
  exit(1);
        Error("Unable to open file.");
}
outfile = fopen("/home/newhall/output.txt", "w");
if (outfile == NULL) {
        Error("Unable to open file.");
}
```

3 USE I/O operations to read, write, or move the current position in the file

```
int ch;
ch = getc(infile);
putc(ch, outfile);
```

4 CLOSE the file: use fclose to close the file after you are done with it

```
fclose(infile);
fclose(outfile);
```

You can also move the current file position in a file:

```
   void rewind(FILE *f);
  rewind(infile);
   fseek(FILE *f, long offset, int whence);
  fseek(f, 0, SEEK_SET);
  fseek(f, 3, SEEK_CUR);
  fseek(f, -3, SEEK_END);
```

Sequential and Random Access File Handling in C

A file handling in C tutorial detailing the use of sequential and random access files in C, along with examples of using the fseek, ftell and rewind functions. In computer programming, the two main types of file handling are:

- Sequential;
- Random access.

Sequential files are generally used in cases where the program processes the data in a sequential fashion – i.e. counting words in a text file – although in some cases, random access can be affected by moving backwards and forwards over a sequential file.

True random access file handling, however, only accesses the file at the point at which the data should be read or written, rather than having to process it sequentially. A hybrid approach is also possible whereby a part of the file is used for sequential access to locate something in the random access portion of the file, in much the same way that a File Allocation Table (FAT) works.

The three main functions that this article will deal with are:

- rewind() – return the file pointer to the beginning;
- fseek() – position the file pointer;
- ftell() – return the current offset of the file pointer.

Each of these functions operates on the C file pointer, which is just the offset from the start of the file, and can be positioned at will. All read/write operations take place at the current position of the file pointer.

The rewind() Function

The rewind() function can be used in sequential or random access C file programming, and simply tells the file system to position the file pointer at the start of the file. Any error flags will also be cleared, and no value is returned.

While useful, the companion function, fseek(), can also be used to reposition the file pointer at will, including the same behavior as rewind().

Using fseek() and ftell() to Process Files

The fseek() function is most useful in random access files where either the record (or block) size is known, or there is an allocation system that denotes the start and end positions of records in an index portion of the file. The fseek() function takes three parameters:

- FILE * f – the file pointer;
- long offset – the position offset;
- int origin – the point from which the offset is applied.

The origin parameter can be one of three values:

- SEEK_SET – from the start;
- SEEK_CUR – from the current position;
- SEEK_END – from the end of the file.

So, the equivalent of rewind() would be:

fseek(f, 0, SEEK_SET);

By a similar token, if the programmer wanted to append a record to the end of the file, the pointer could be repositioned thus:

fseek(f, 0, SEEK_END);

Since fseek() returns an error code (0 for no error) the stdio library also provides a function that can be called to find out the current offset within the file:

long offset = ftell(FILE * f)

This enables the programmer to create a simple file marker (before updating a record for example), by storing the file position in a variable, and then supplying it to a call to fseek:

```
long file_marker = ftell(f);
// … file processing functions
fseek( f, file_marker, SEEK_SET);
```

Of course, if the programmer knows the size of each record or block, arithmetic can be used. For example, to rewind to the start of the current record, a function call such as the following would suffice:

fseek(f, 0 – record_size, SEEK_CURR);

With these three functions, the C programmer can manipulate both sequential and random access files, but should always remember that positioning the file pointer is absolute. In other words, if fseek is used to position the pointer in a read/write file, then writing will overwrite existing data, permanently.