CSC-01 INTRODUCTION TO COMPUTING

Introductory Concepts

C Fundamentals

Program Structure

Preparing and Running a Complete C Program

Operators and Expressions

Data Input and Output

Control Statements

Structure and Unions

Arrays

Pointers

Functions

Data Files

Low-Level Programming.

Introductory Concepts

COMPUTER? An electronic device which is capable of receiving information (data) in a particular form and of performing a sequence of operations in accordance with a predetermined but variable set of procedural instructions (program) to produce a result in the form of information or signals.
COMPUTING? the use or operation of computers
Use of a computer? Word Processing, Web Surfing, Instant Messaging / Email, Music, Movies, Games, Air traffic control, Car diagnostics, Climate control
Parts of Computer Processor *brains,* Memory *scratch paper,* Disk *long term memory,* I/O *communication (senses),* Software *reconfigurability*

Types of Computers:

Microcomputers (personal computers) single chip microprocessors
Desktop computers - A case and a display put on a desk.
Laptops and notebook computers – Portable and all in two foldable cases.
Tablet computer – Portable and all in one case.
Smartphones – Small handheld computers with limited hardware.

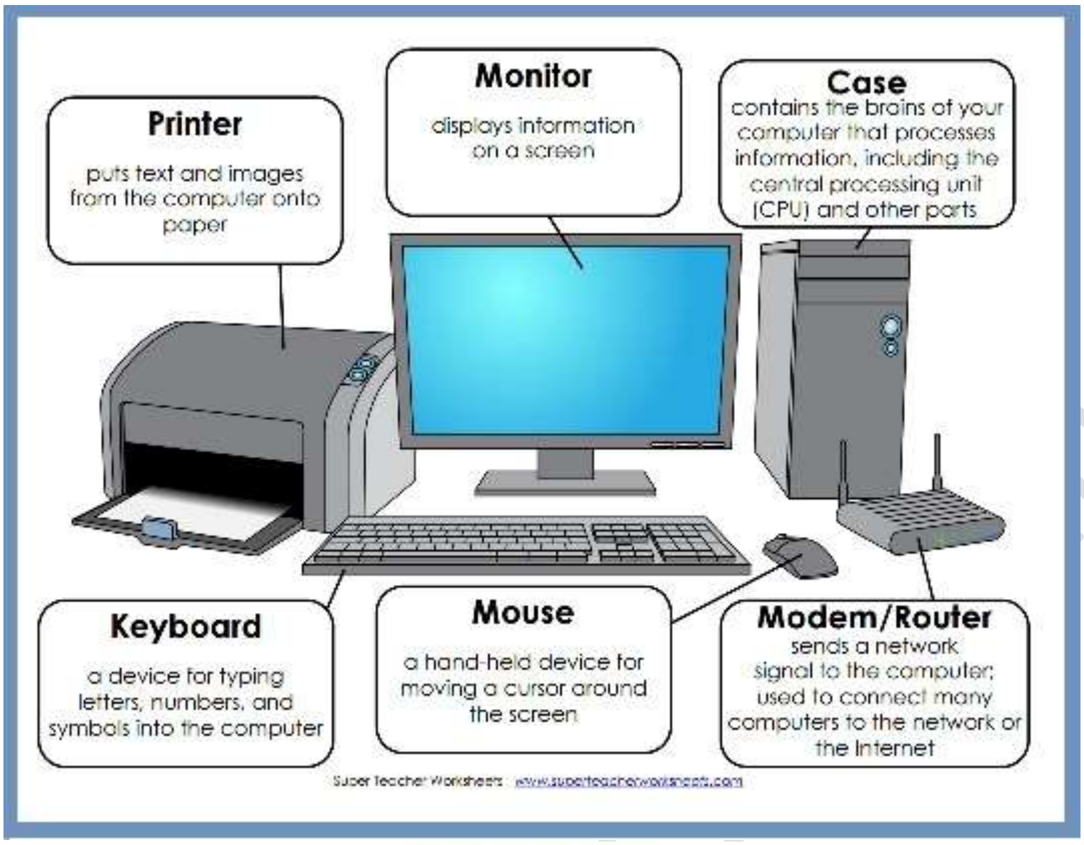
Servers
Server usually refers to a computer that is dedicated to provide a service. For example, a computer dedicated to a database may be called a "database server". "File servers" manage a large collection of computer files. "Web servers" process web pages and web applications. Many smaller servers are actually personal computers that have been dedicated to provide services for other computers.

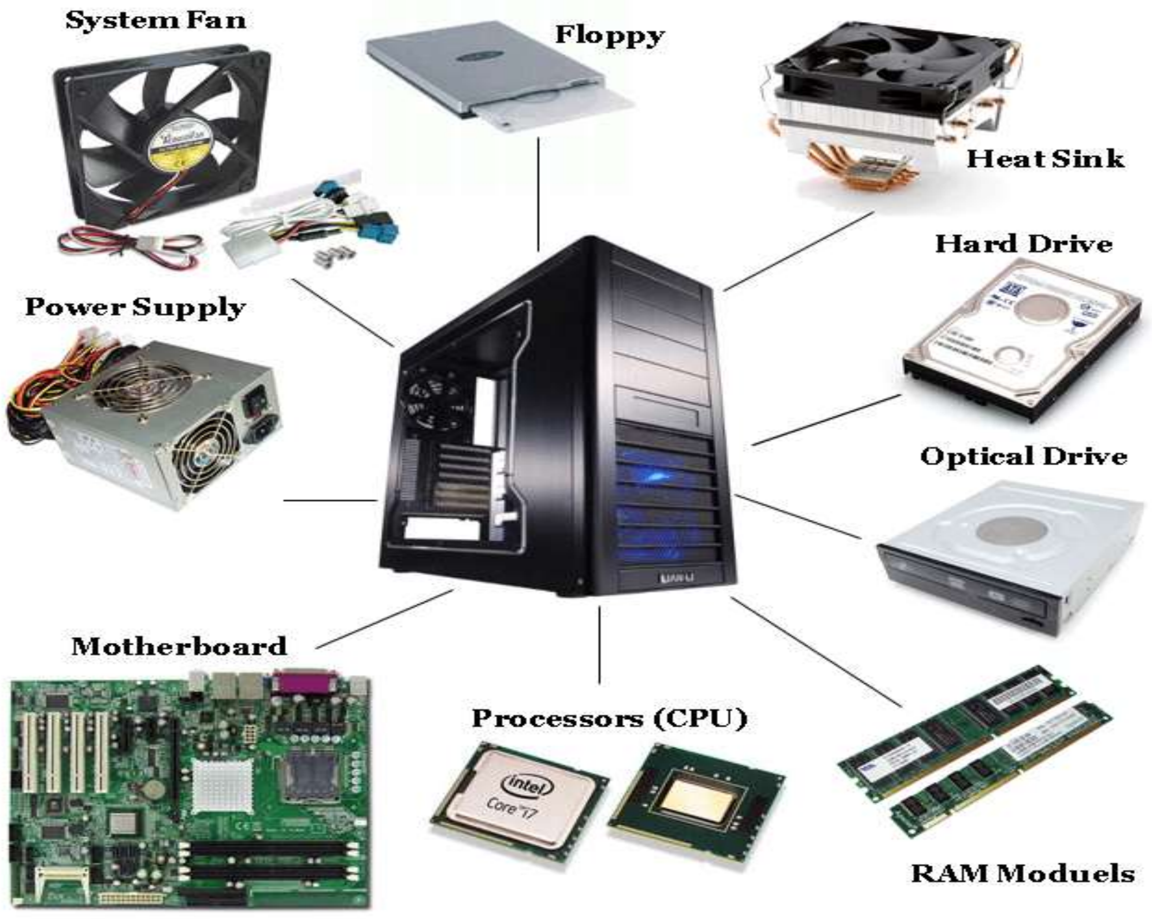
Workstations
Workstations are computers that are intended to serve one user and may contain special hardware enhancements not found on a personal computer. By the mid 1990s personal computers reached the processing capabilities of Mini computers and Workstations. Also, with the release of multi-taskingsystems such as OS/2, Windows NT and Linux, the operating systems of personal computers could do the job of this class of machines.

Minicomputers (midrange computers)
Minicomputers are a class of multi-user computers that lie in the middle range of the computing spectrum, in between the smallest mainframe computers and the largest single-user systems (microcomputers or personal computers).

Mainframe computers
The term mainframe computer was created to distinguish the traditional, large, institutional computer intended to service multiple users from the smaller, single user machines. They are measured in MIPS (million instructions per second) and respond to up to 100s of millions of users at a time.

Supercomputers
A Supercomputer is focused on performing tasks involving intense numerical calculations such as weather forecasting, fluid dynamics, nuclear simulations, theoretical astrophysics, and complex scientific computations. Supercomputer processing speeds are measured in floating point operations per second, or FLOPS.


Printer
puts text and images from the computer onto paper
Monitor
displays information on a screen
Case
contains the brains of your computer that processes information, including the central processing unit (CPU) and other parts
Keyboard
a device for typing letters, numbers, and symbols into the computer
Mouse
a hand-held device for moving a cursor around the screen
Modem/Router
sends a network signal to the computer; used to connect many computers to the network or the Internet
Super Teacher Worksheets www.superteacherworksheets.com


System Fan
Floppy
Heat Sink
Hard Drive
Power Supply
Optical Drive
Motherboard
Processors (CPU)
RAM Moduels

Webcam
Webcam
Speakers
Microphone
Monitor
Printer
Printer
Headphone
Memory Card Reader
USB Flash Memory
Media Devices
External Optical Drives
ZIP Drive

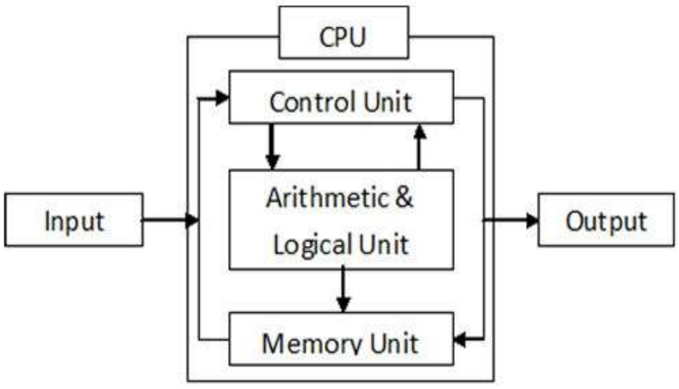


Fig. Block Diagram of Computer

1. Input unit – Input unit is a unit that accepts any input device. The input device is used to input data into the computer system.

Function of input unit: It converts inputted data into binary codes. It sends data to main memory.

2. Central Processing Unit (CPU) – CPU is called the brain of a computer. An electronic circuitry that carries out the instruction given by a computer program. CPU can be sub classified into three parts.

a. Control unit (CU) b. Arithmetic & Logic unit (ALU) c. Memory Unit (MU)

a. Control unit (CU) - the control unit manages the various components of the computer. It reads instructions from memory and interpretation and changes in a series of signals to activate other parts of the computer. It controls and co-ordinate is input output memory and all other units.

b. Arithmetic & Logic unit (ALU) – The arithmetic logic unit (ALU), which performs simple arithmetic operation such as +,-, *, / and logical operation such as >, <, =<, <= etc.

c. Memory Unit (MU) - Memory is used to store data and instructions before and after processing. Memory is also called Primary memory or internal memory. It is used to store data temporary or permanently.

Function of CPU-

It controls all the parts and software and data flow of computer.
It performs all operations.
It accepts data from input device.
It sends information to output device.
Executing programs stored in memory
It stores data either temporarily or permanent basis.
It performs arithmetical and logical operations.

3. Output Unit –Output unit is a unit that constituents a number of output device. An output device is used to show the result of processing.

Function of Output unit: It accepts data or information sends from main memory of computer
It converts binary coded information into HLL or inputted languages.

Cabinet: SMPS 450, 500, 550, 600, 650, 750, 850, 900 WATT

Mother Board: Intel, Zebronics, Asus, Gigabyte, Mercury, Frontech, Msi, Ecs, Chipset, Memoryslots, Form Factor (size, configuration)

Processors: AMD A10-5800K. AMD FX-9590. AMD Sempron 3850, Intel Core i3-6100. Intel Core i7-6700K. Intel Core i5-4690K, CPU Cores 4, 8, GPU Cores, CPU Frequency, 3M, 6M, 8M, Cache (L2), 3.00GHz, 3.1GHz, 3.2GHz, 3.4GHz, 3.5GHz, 3.9GHz, 4.0GHz, 4.2GHz, 4.40GHz, first, second, third, fourth generation.

ROM / RAM: BIOS, DDR1, DDR2, DDR3, DDR4, 2GB, 4GB, 8GB.

HDD: HP, Kingston, ADATA, Sony, Toshiba, Seagate, WD, Barracuda – SSD, SATA, SCSI, 5400RPM, 7200RPM, 10,000RPM, 500GB, 1TB, 2TB, 3TB.

BD, DVD, CD: HP, LG, Sony, Samsung. 8x, 18x, 24x, 48x, SATA, RW, Cache, Form Factor 5.25”.

USB – 2.0, 3.0, 3.1, Keyboard – QWERTY, Mouse – Optical mouse, left, right click, scroll

Monitors – CRT (**Cathode Ray Tube)**, TFT, LCD (Liquid crystal display), LED (Light Emitting Diode), OLED (Organic LED), Plasma. ***Screen Size (***13, 15, 17, 19, 21, 23, 27 and 32 inches***), Viewing Angle (***The viewing angle indicates at what angle the monitor can be viewed vertically and horizontally and still be seen***.), Contrast Ratio (***The contrast ratio determines how rich colors will appear on-screen, the higher the ratio the better. Contrast ratios range from 200:1 up to 1000:1***), Resolution and Refresh Rates.*** The resolution is the number of dots displayed on the entire screen. The higher the resolution the smaller everything on the screen will be. This can be a benefit for running multiple applications at the same time but can also be a burden for someone with poor eyesight. Common resolution supports include 640 * 480, 800 * 600 and 1,024*768 and so forth. The refresh rate of a monitor is the frequency at which the screen is redrawn. The higher the number the more often the screen is redrawn and the less flicker will occur. Common Refresh rate are 60 to 80Hertz.

Printers: Dot matrix (80/132 cloumn), Inkjet, Laserjet, A8 (Business card-2.07”x2.91”) /A7…A4(8.27”x11.69”)/A3/A0/2A Plotter, Black & White, Colour.

Speakers – Mono, Stereo, Home Theatre, 2.1, 5.1, 7.1, SubWoofer, Wireless, Bluetooth.

System Software: Dedicated to managing the computer itself, such as the operating system, file management utilities, and disk operating system. The computer programs used to start and run computer systems.

Application Software: Specific purpose programs word processing, web browsers, accounting, truck scheduling, Astrology, Music Player, Movie Player, Video Games, etc.

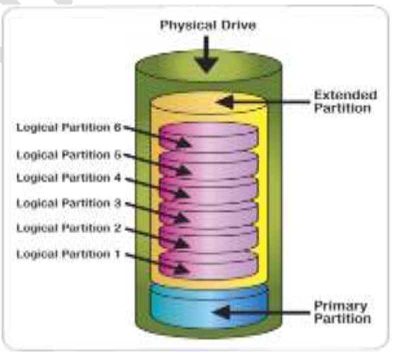
Programming Languages (High / Low Level): Language is set of instructions to perform specific task. High level languages use common simple English words for instructions. No need of detailed information about computer hardware. Example: C, C++, Java, etc. Low level languages use specific symbols for instructions. Detailed information is required about computer hardware. Example: Assembly Language.

Compiler: is a computer program (or a set of programs) that transforms source code (Program) written in a programming language (the source language) into another computer language / object code (binary language).

Interpreter: is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program.

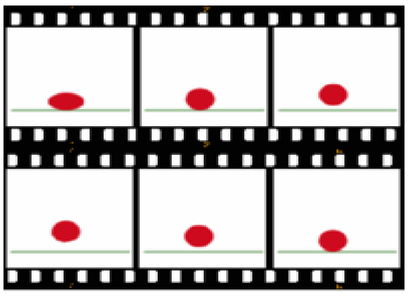
Booting: is the initialization of a computerized system. The booting process can be "hard", after electrical power to the CPU is switched from off to on (in order to diagnose particular hardware errors), or "soft", when those power-on self-tests (POST) can be avoided. A boot loader is a computer program that loads an operating system or some other system software for the computer after completion of the power-on self-tests; it is the loader for the operating system itself. Within the hard reboot process, it runs after completion of the self-tests, then loads and runs the software. A boot loader is loaded into main memory from persistent memory, such as a hard disk drive.

Managing disk partitions and File System: Partitioning is typically the first step of preparing a newly manufactured disk, before any files or directories have been created. The disk stores the information about the partitions' locations and sizes in an area known as the partition table that the operating system reads before any other part of the disk. Each partition then appears in the operating system as a distinct "logical" disk that uses part of the actual disk. System administrators use a program called a partition editor to create, resize, delete, and manipulate the partitions.



In windows, Go to Control Panel, Disk Management for managing partitions.

Multimedia is content that uses a combination of different content forms such as text, audio, images, animation, video and interactive content. Graphics are visual images or designs on some surface, such as a wall, canvas, screen, paper, or stone to inform, illustrate, or entertain. In contemporary usage it includes: pictorial representation of data, as in computer-aided design and manufacture, in typesetting and the graphic arts, and in educational and recreational software. Images that are generated by a computer are called computer graphics. Applications: cinema presentation, video game, simulator, etc. Animation is the process of making the illusion of motion and change by means of the rapid display of a sequence of static images that minimally differ from each other. Images are displayed in a rapid succession, usually 24, 25, 30, or 60 frames per second.



Number system

Computers can understand only numbers. Thus, all data such as the words, images, music, etc. are translated by the computer to numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.
The value of each digit in a number can be determined using −
The digit
The position of the digit in the number
The base of the number system (where the base is defined as the total number of digits available in the number system)

Decimal Number System
The number system that we use in our day-to-day life is the decimal number system. Decimal number system has base 10 as it uses 10 digits from 0 to 9. In decimal number system, the successive positions to the left of the decimal point represent units, tens, hundreds, thousands, and so on.
Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position. Its value can be written as

```
(1 x 1000)+ (2 x 100)+ (3 x 10)+ (4 x l)
(1 x 10^3)+ (2 x 10^2)+ (3 x 10^1)+ (4 x l0^0)
1000 + 200 + 30 + 4
1234
```

As a computer programmer or an IT professional, you should understand the following number systems which are frequently used in computers.

Binary Number System
Characteristics of the binary number system are as follows −
Uses two digits, 0 and 1, called as base 2 number system
Each position in a binary number represents a 0 power of the base (2). Example 2^0
Last position in a binary number represents a x power of the base (2).
Example 2^x where x represents the last position - 1.
Example
Binary Number: 10101_2
Calculating Decimal Equivalent –

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \text{ x } 2^4) + (0 \text{ x } 2^3) + (1 \text{ x } 2^2) + (0 \text{ x } 2^1) + (1 \text{ x } 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$
Step 3	10101_2	21_{10}

Octal Number System
Characteristics of the octal number system are as follows −
Uses eight digits, 0, 1, 2, 3, 4, 5, 6, 7
Also called as base 8 number system
Each position in an octal number represents a 0 power of the base (8). Example 8^0

Last position in an octal number represents a x power of the base (8). Example
8^x where x represents the last position - 1
Example Octal Number: 12570_8
Calculating Decimal Equivalent −

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

Hexadecimal Number System

Characteristics of hexadecimal number system are as follows −
Uses 10 digits and 6 letters, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
Letters represent the numbers starting from 10. A = 10. B = 11, C = 12, D = 13, E = 14, F = 15
Also called as base 16 number system
Each position in a hexadecimal number represents a 0 power of the base (16). Example, 160
Last position in a hexadecimal number represents a x power of the base (16). Example 16x where x represents the last position - 1
Example Hexadecimal Number: $19FDE_{16}$
Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536+ 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

Decimal to Other Base System

Step 1 − Divide the decimal number to be converted by the value of the new base.
Step 2 − Get the remainder from Step 1 as the rightmost digit (least significant digit) of the new base number.
Step 3 − Divide the quotient of the previous divide by the new base.
Step 4 − Record the remainder from Step 3 as the next digit (to the left) of the new base number.
Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.
The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.
Example Decimal Number: 29_{10}
Calculating Binary Equivalent −

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).

Decimal Number: 29_{10} = Binary Number: 11101_{2}.

Other Base System to Non-Decimal System
Step 1 − Convert the original number to a decimal number (base 10).
Step 2 − Convert the decimal number so obtained to the new base number.
Example
Octal Number: 25_8
Calculating Binary Equivalent −
Step 1 - Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number: 25_8 = Decimal Number: 21_{10}

Step 2 - Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0
Step 5	1 / 2	0	1

Decimal Number: 21_{10} = Binary Number: 10101_2

Octal Number: 25_8 = Binary Number: 10101_2

Shortcut Method ─ Binary to Octal
Step 1 − Divide the binary digits into groups of three (starting from the right).
Step 2 − Convert each group of three binary digits to one octal digit.
Example Binary Number: 10101_2
Calculating Octal Equivalent −

Step	Binary Number	Octal Number
Step 1	10101_2	010 101
Step 2	10101_2	2_8 5_8
Step 3	10101_2	25_8

Binary Number: 10101_2 = Octal Number: 25_8

Shortcut Method ─ Octal to Binary
Step 1 − Convert each octal digit to a 3-digit binary number (the octal digits may be treated as decimal for this conversion).
Step 2 − Combine all the resulting binary groups (of 3 digits each) into a single binary number.
Example Octal Number: 25_8
Calculating Binary Equivalent −

Step	Octal Number	Binary Number
Step 1	25_8	2_{10} 5_{10}
Step 2	25_8	010_2 101_2
Step 3	25_8	010101_2

Octal Number: 25_8 = Binary Number: 10101_2

Shortcut Method ─ Binary to Hexadecimal
Step 1 − Divide the binary digits into groups of four (starting from the right).
Step 2 − Convert each group of four binary digits to one hexadecimal symbol.
Example Binary Number: 10101_2
Calculating hexadecimal Equivalent −

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101
Step 2	10101_2	1_{10} 5_{10}
Step 3	10101_2	15_{16}

Binary Number: 10101_2 = Hexadecimal Number: 15_{16}

Shortcut Method - Hexadecimal to Binary
Step 1 − Convert each hexadecimal digit to a 4-digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
Step 2 − Combine all the resulting binary groups (of 4 digits each) into a single binary number.
Example Hexadecimal Number: 15_{16}
Calculating Binary Equivalent −

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	1_{10} 5_{10}
Step 2	15_{16}	0001_2 0101_2
Step 3	15_{16}	00010101_2

Hexadecimal Number: 15_{16} = Binary Number: 10101_2

Convert decimal fraction to binary

- Multiply the fractional decimal number by 2.
- Integral part of resultant decimal number will be first digit of fraction binary number.
- Repeat step 1 using only fractional part of decimal number and then step 2.

$0.47_{10} * 2_{10} = 0.94_{10}$, Integral part: 0_2
$0.94_{10} * 2_{10} = 1.88_{10}$, Integral part: 1_2
$0.88_{10} * 2_{10} = 1.76_{10}$, Integral part: 1_2
$0.47_{10} = 0.011_2$

Convert binary fraction to decimal

- Divide each digit from right side of radix point till the end by 2^1, 2^2, 2^3, … respectively.
- Add all the result coming from step 1.
- Equivalent fractional decimal number would be the result obtained in step 2.

=> $0.101_2 = (1*1/2) + (0*1/2^2) + (1*1/2^3)$
=> $0.101_2 = 1*0.5 + 0*0.25 + 1*0.125$
=> $0.101_2 = 0.625_{10}$

Convert decimal fraction to hexa

- multiply 0.00390625 by 16 and take the integer part

0.00390625 x 16 = 0.0625,
Integer part = 0,
Fractional part = 0.0625

- multiply 0.0625 by 16 and take the integer part

0.0625 x 16 = 1.000
Integer part = 1
Fractional part = 0

$0.00390625_{10} = 0.01_{16}$

Convert hexa fraction to decimal

- 0.16_{16}
- multiply 1 by 1/16 and take the integer part

1 x 1/16 = 0.625

- multiply 6 by 16 and take the integer part

6 x $1/16^2$ = 0.023

$0.16_{16} = 0.648_{10}$

Convert decimal fraction to octal

- multiply 0.015625 by 8 and take the integer part

0.015625 x 8 = 0.125
Integer part = 0
Fractional part = 0.125

- multiply 0.125 by 8 and take the integer part

0.125 x 8 = 1.000
Integer part = 1
Fractional part = 0

$0.015625_{10} = 0.01_8$

Convert octal fraction to decimal

- multiply 0.16 by 8 and take the integer part

0.16 x 8 = 1.28
Integer part = 1
Fractional part = 0.28

- multiply 0.28 by 8 and take the integer part

0.28 x 8 = 2.24
Integer part = 2
Fractional part = 0.24

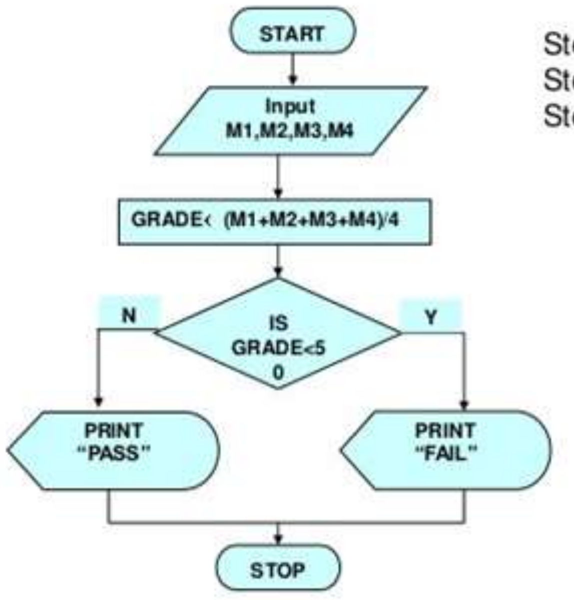
$0.16_{10} = 0.12_8$

Algorithm

An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

Flowchart

A flowchart is a type of diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution model to a given problem.



```
Step 1:  Input M1,M2,M3,M4
Step 2:  GRADE ← (M1+M2+M3+M4)/4
Step 3:  if (GRADE <50) then
             Print "FAIL"
          else
             Print "PASS"
         endif
```

C Language Introduction

C is a high-level structured oriented programming language, used in general purpose programming, developed by Dennis Ritchie at AT&T Bell Labs, the USA between 1969 and 1973.

In 1988, the American National Standards Institute (ANSI) had formalized the C language. C was invented to write UNIX operating system. C is a successor of 'Basic Combined Programming Language' (BCPL) called B language. Linux OS, PHP, and MySQL are written in C. C has been written in assembly language.

In the beginning, C was used for developing system applications, e.g.:

Database Systems, Language Interpreters, Compilers and Assemblers, Operating Systems, Network Drivers, Word Processors.

Fundamentals

The basic elements used to construct a simple C program are: the C character set, identifiers and keywords, data types, constants, arrays, declarations, expressions and statements. Let us see how these elements can be combined to form more comprehensive program components-

Low-Level Programming

Object code is small and efficient. Optimize the use of three resources: Execution time, Memory, Development/maintenance time.

The C Character Set:

C uses letters A to Z in lowercase and uppercase, the digits 0 to 9, certain special characters, and white spaces to form basic program elements (e.g variables, constants, expressions etc.) The special characters are:

+ – * / = % & # ! ? ^ " ' / | < > ( ) [ ] { } : ; . , ~ @ !

The white spaces used in C programs are: blank space, horizontal tab, carriage return, new line and form feed.

Identifiers and Keywords:

Identifiers are names given to various program elements such as variables, functions, and arrays. Identifiers consist of letters and digits, in any order, except that the first character must be a letter. Both uppercase and lowercase letters are permitted and the underscore may also be used, as it is also regarded as a letter. Uppercase and lowercase letters are not equivalent, thus not interchangeable. This is why it is said that C is case sensitive. An identifier can be arbitrarily long.
The same identifier may denote different entities in the same program, for example, a variable and an array may be denoted by the same identifier, example below.

int sum, average, A[10]; // sum, average and the array name A are all identifiers.

The _ _func_ _ predefined identifier:-

The predefined identifier __func__ makes a function name available for use within the function. Immediately following the opening brace of each function definition, _ _func_ _ is implicitly declared by the compiler in the following way:

```
static const char _ _func_ _[] = "function-name";
```
where function-name is the name of the function.

consider the following example

```
#include <stdio.h>
void myfunc(void)    {
          printf("%s",__func__);
          return;
}
void main() {
     myfunc();
}
```
The output would be   myfunc

Keywords

Keywords are reserved words that have standard predefined meanings. These keywords can only be used for their intended purpose; they cannot be used as programmer defined identifiers. Examples of some keywords are: int, main, void, if.

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

Data Types

Data values passed in a program may be of different types. Each of these data types are represented differently within the computer's memory and have different memory requirements. These data types can be augmented by the use of data type qualifiers / modifiers.

The data types supported in C are described below:

int:

It is used to store an integer quantity. An ordinary int can store a range of values from INT_MIN to INT_MAX as defined by in header file <limits.h>. The type modifiers for the int data type are: signed, unsigned, short, long, and long long.

- A short int occupies 2 bytes of space and a long int occupies 4 bytes.
- A short unsigned int occupies 2 bytes of space but it can store only positive values in the range of 0 to 65535.
- An unsigned int has the same memory requirements as a short unsigned int. However, in case of an ordinary int, the leftmost bit is reserved for the sign.
- A long unsigned int occupies 4 bytes of memory and stores positive integers in the range of 0 to 4294967295.
- By default the int data type is signed.
- A long long int occupies 64 bits of memory. It may be signed or unsigned. The signed long, long int stores values from −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and the unsigned long long ranges from 0 to 18,446,744,073,709,551,615.

char:

It stores a single character of data belonging to the C character set. It occupies 1 byte of memory, and stores any value from the C character set. The type modifiers for char are signed and unsigned.
Both signed and unsigned char occupy 1 byte of memory but the range of values differs. An unsigned char can store values from 0 to 255 and a signed char can store values from -128 to +127. Each char type has an equivalent integer interpretation, so that a char is really a special kind of short integer. By default, char is unsigned.

float:

It is used to store real numbers with single precision i.e. a precision of 6 digits after decimal point. It occupies 4 bytes of memory. The type modifier for float is long. It has the same memory requirements as double.

double:

It is used to store real numbers with double precision. It occupies 8 bytes of memory. The type modifier for double is long. A long double occupies 10 bytes of memory.

void:
It is used to specify an empty set containing no values. Hence, it occupies 0 bytes of memory.

_Bool:
A boolean data type, which is an unsigned integer type, that can store only two values, 0 and 1. Include the file <stdbool.h> when using _Bool.

_Complex:

It is used to store complex numbers. There are three complex types: float _Complex, double _Complex, and long double _ComplexIt is found in the <complex.h> file.

Constants:

A constant is an identifier whose value remains unchanged throughout the program. To declare any constant the syntax is: const datatype varname = value; where const is a keyword that declares the variable to be a fixed value entity.
There are four basic types of constants in C. They are integer constants, floating point constants, character constants and string constants. Integer and floating point constants cannot contain commas or blank spaces; but they can be prefixed by a minus sign to indicate a negative quantity.

Integer Constants:

An integer constant is an integer valued number. It consists of a sequence of digits. Integer constants can be written in the following three number systems:
Decimal (base 10): A decimal constant can consist of any combination of digits from 0 to 9. If it contains two or more digits, the first digit must be something other than 0, for example: const int size =50;

Octal (base 8): An octal constant can consist of any combination of digits from 0 to 7. The first digit must be a 0 to identify the constant as an octal number, for example: const int a= 074; const int b= 0;

Hexadecimal constant (base 16): A hexadecimal constant can consist of any combination of digits from 0 to 9 and a to f (either uppercase or lowercase). It must begin with 0x or oX to identify the constant as a hexadecimal number, for example: const int c= 0x7FF;

Integer constants can also be prefixed by the type modifiers unsigned and long. Unsigned constants must end with uor U, long integer constants must end with lor L and unsigned long integer constants must end with ul or UL. Long long integer constants end with LL or ll. unsigned long long end with UL or ul

Floating Point Constant:

Its a base 10 or a base 16 number that contains a decimal point or an exponent or both. In case of a decimal floating point constant the exponent the base 10 is replaced by e or E. Thus, 1.4 *10^-3 would be written as 1.4E-3 or 1.4e-3.

In case of a hexadecimal character constant, the exponent is in binary and is replaced by p or P. For example:

```
const float a= 5000. ;

const float b= .1212e12;

const float c= 827.54;
```

Floating point constants are generally double precision quantities that occupy 8 bytes. In some versions of C, the constant is appended by F to indicate single precision and by L to indicate a long floating point constant.

Character Constants:

A character constant is a sequence of one or more characters enclosed in apostrophes. Each character constant has an equivalent integer value that is determined by the computer's character set. It may also contain escape sequences. A character literal may be prefixed with the letter L, u or U, for example L'c'.

A character literal without the L prefix is an ordinary character constant or a narrow character constant. A character literal with the L prefix is a wide character constant. The type of a narrow character constant and a multicharacter constant is int. The type of a wide character constant with prefix L is wchar_t defined in the header file <stddef.h> .A wide character constant with prefix u or U is of type char16_t or char32_t. These are unsigned character types defined in <uchar.h>.

An ordinary character literal that contains more than one character or escape sequence is a multicharacter constant, for example: const char p= 'A';

Escape Sequences are also character constants that are used to express certain non printing characters such as the tab or the carriage return.An escape sequence always begins with a backward slash and is followed by one or more special characters. for eg. b will represent the bell, n will represent the line feed.

String Literals:

A string literal consists of a sequence of multibyte characters enclosed in double quotation marks. They are of two types, wide string literal and UTF-8 string literal. A UTF-8 string literal is prefixed by u8 and a wide string literal by L, uor U.

The compiler recognizes and supports the additional characters (the extended character set) which you can meaningfully use in string literals and character constants. The support for extended characters includes the multibyte character sets. A multibyte character is a character whose bit representation fits into one or more bytes.

Symbolic Constants:

A symbolic constant is a name that substitutes for a numeric constant, a character constant or a string constant throughout the program. When the program is compiled each occurrence of a symbolic constant is replaced by its actual value.

A symbolic constant is defined at the beginning of a program using the # **define** feature. The # define feature is called a preprocessor directive, more about the C preprocessor in a later article.

A symbolic constant definition never ends with a semi colon as it is not a C statement rather it is a directive, for example:

```
#define  PI  3.1415    //PI is the constant that will
represent value 3.1415

#define  True 1

#define  name  "Alice"
```

Program Structure

C program basically consists of the following parts −

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World" −

```
#include <stdio.h>
int main() {
   /* my first program in C */
   printf("Hello, World! \n");
   return 0;
}
```

Let us take a look at the various parts of the above program –

The first line of the program #include <stdio.h> is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.

The next line int main() is the main function where the program execution begins.

The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

The next line printf(...) is another function available in C which causes the message "Hello, World!" to be displayed on the screen.

The next line return 0; terminates the main() function and returns the value 0.

Preparing and Running a Complete C Program

Operators and Expressions

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

We will, in this chapter, look into the way each operator works.

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

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

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

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

>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

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

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

Bitwise Operators

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

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

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

2’s complement of N = -(N+1)

Example, 2’s complement of 35 = -36, 2’s complement of -12 = 11

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

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

```
#include <stdio.h>
main() {
   unsigned int a = 60;     /* 60 = 0011 1100 */
   unsigned int b = 13;     /* 13 = 0000 1101 */
   int c = 0;
   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );
   c = a | b;       /* 61 = 0011 1101 */
   printf("Line 2 - Value of c is %d\n", c );
   c = a ^ b;       /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );
   c = ~a;          /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );
   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );
   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );
}
```

Assignment Operators

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

Operator	Description	Example
=	Simple assignment operator. Assigns values from	C = A + B will assign the

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

Misc Operators ↦ sizeof & ternary
Besides the operators discussed above, there are a few other important operators including sizeof and ? : supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;

? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Data Input and Output

When we say Input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say Output, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The int getchar(void) function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The int putchar(int c) function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example −

```
#include <stdio.h>
int main( ) {
   int c;
   printf( "Enter a value :");
   c = getchar( );
   printf( "\nYou entered: ");
   putchar( c );
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

```
$./a.out
Enter a value : this is test
You entered: t
```

The gets() and puts() Functions

The char *gets(char *s) function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF (End of File).

The int puts(const char *s) function writes the string 's' and 'a' trailing newline to stdout.

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

```
int main( ) {

   char str[100];

   printf( "Enter a value :");

   gets( str );

   printf( "\nYou entered: ");

   puts( str );

   return 0;

}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

```
$./a.out
Enter a value : this is test
You entered: this is test
```

The scanf() and printf() Functions

The int scanf(const char *format, ...) function reads the input from the standard input stream stdin and scans that input according to the formatprovided.

The int printf(const char *format, ...) function writes the output to the standard output stream stdout and produces the output according to the format provided.

The format can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better −

```
#include <stdio.h>

int main( ) {

   char str[100];

   int i;

   printf( "Enter a value :");

   scanf("%s%d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;

}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows −

```
$./a.out
Enter a value: seven 7
You entered: seven 7
```

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

```
int a;
scanf("%5d", &a);
```

take just first 5 digit as input. Any digits after 5th one would be ignored.

```
char str[100];
scanf("%[^\n]", str);
%[0–9]
```

Control Statements

C provides two styles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:
Branching is so called because the program chooses to follow one branch or another.

if statement

This is the simplest form of the branching statements.

It takes an expression in parenthesis and a statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaulated values is non-zero.

if statements take the following form:

```
if (expression)
  statement;

or

if (expression)  {
    Block of statements;
  }

or

if (expression)  {
    Block of statements;
  }
else  {
    Block of statements;
  }

or

if (expression)  {
    Block of statements;
  }
```

```
else if(expression)  {
    Block of statements;
  }
else  {
    Block of statements;
  }
```

? : Operator
The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.
? : is a ternary operator in that it takes three values, this is the only ternary operator C has.
? : takes the following form:

if condition is true ? then X return value : otherwise Y value;

switch statement:
The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

```
switch( expression )
     {
        case constant-expression1:    statements1;
        [case constant-expression2:   statements2;]
        [case constant-expression3:   statements3;]
        [default : statements4;]
     }
```

Using break keyword:
If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using *break* keyword.
What is default condition:
If none of the listed conditions is met then default condition executed.
Looping
Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.
while loop
The most basic loop in C is the while loop. A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statements get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.
Basic syntax of while loop is as follows:

```
while ( expression )
{
   Single statement
   or
   Block of statements;
}
```

for loop
for loop is similar to while, it's just written differently. for statements are often used to proccess lists such a range of numbers:
Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
   Single statement
   or
   Block of statements;
}
```

In the above syntax:

- expression1 - Initialisese variables.
- expression2 - Condtional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

```
do
{
   Single statement
   or
   Block of statements;
}while(expression);
```

break and continue statements

C provides two commands to control how we loop:

- break -- exit form loop or switch.
- continue -- skip 1 iteration of loop.

You already have seen example of using break statement. Here is an example showing usage of **continue** statement.

```
#include

main()
{
    int i;
    int j = 10;
    for( i = 0; i <= j; i ++ )    {
       if( i == 5 )       {
          continue;
       }
       printf("Hello %d\n", i );
    }
}
```

This will produce following output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
```

Function

- input
- process
- output

```
return-type function_name (parameter-list){
      statements ;
      return value;
}
```

Example

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

Calling Functions

```
function-name (parameters);
add_values (5,4);
int a=3,b=6;
add_values (a, b);
printf("%d", add_values (5,4));
```

Function Parameters

A literal value or a value stored in variable or an address in memory or a more complex expression built by combining these

```
int foo (int a) { a = 2 * a; return a; }
x = foo (x);
void foo (int *x) { *x = *x + 42; } int a = 15; foo (&a);
```

The main Function

- Every program requires at least one function, called 'main'. This is where the program begins executing.
- You do not need to write a declaration or prototype for main, but you do need to define it.
- The return type for main is always int. You do not have to specify the return type for main, but you can.
- However, you *cannot* specify that it has a return type other than int.
- the return value from main indicates the program's *exit status*.
- A value of zero or EXIT_SUCCESS indicates success and EXIT_FAILURE indicates an error.
- Reaching the } at the end of main without a return, or executing a return statement with no value (that is, return;) are both equivalent.

```
void main (void) {
      puts ("Hi there!");
}
```

Example

```
int main (int argc, char *argv[]) {
      int counter;
      for (counter = 0; counter < argc; counter++)
      printf ("%s\n", argv[counter]);
      return 0;
}
```

Recursive Functions a function that calls itself

```
int factorial (int x) {
      if (x < 1) return 1;
      else return (x * factorial (x - 1));
}
```

Infinitely Recursive

```
int watermelon (int x) {
      return (watermelon (x));
}
```

Static Functions

Callable only within the source file where it is defined. Useful for building a reusable library of functions and need to include some subroutines that should not be callable by the end user.

```
static int foo (int x) {
      return x + 42;
}
```

Nested Functions can define functions within other functions

```
int factorial (int x) {
      int factorial_helper (int a, int b)
      {
            if (a < 1) { return b; }
            else { return factorial_helper ((a - 1), (a * b)); }
      }
      return factorial_helper (x, 1);
}

// main.c
#include "add.c"
int main(void) {
    int result = add(5,6);
    printf("%d\n", result);
}
// add.c
int add(int a, int b) {
    return a + b;
}
```

1	**Call by value** This method passes the actual value as an argument to the function. In this case, changes made inside the function have no effect on the calling function.
2	**Call by reference** This method passes the address to the function. Inside the function, the address is used to access the actual argument used in the call. This effects the variables in the calling function.

Way-1 Formal parameters as a pointer –

```
void myFunction(int *param) {

}
```

Way-2 Formal parameters as a sized array –

```
void myFunction(int param[10]) {

}
```

Way-3 Formal parameters as an unsized array –

```
void myFunction(int param[]) {

}
```

Example

```
double getAverage(int arr[], int size) {
   int i;
   double avg;
   double sum = 0;
   for (i = 0; i < size; ++i)      sum += arr[i];
   avg = sum / size;
   return avg;
}
#include <stdio.h>
double getAverage(int arr[], int size);
void main () {
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   avg = getAverage(balance,5);
   printf("Average value is: %f ",avg);
}
```

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
void main(){
    int num[2][2], i, j;
    printf("Enter 4 numbers:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j) scanf("%d", &num[i][j]);
    displayNumbers(num);
}
void displayNumbers(int num[2][2])
{
    int i, j;
    for (i = 0; i < 2; ++i){
        for (j = 0; j < 2; ++j) printf("%d\t", num[i][j]);
        printf("\n");}
}
```

Storage Class

A storage class defines the scope and life-time of variables and/or functions. Four different storage classes are `auto, register, static, extern.`

The **auto** storage class is the default storage class.

```
int mount;    auto int month;
```

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). The register should only be used for variables that require quick access such as counters.

```
register int  miles;
```

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

```
#include <stdio.h>
void func( void ) {
   static int i = 5;        /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
static int count = 5; /* global variable */
void main(){
   while(count--)func();
}
```

The extern storage class is used to give a reference of a **global variable** or **function** that is visible to **ALL the program files**. When you use 'extern', the variable **cannot be initialized** however, it points the variable name at a storage location that has been previously defined. When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>
 int count ;
extern void write_extern();
 main() {
   count = 5;
   write_extern();
}
```

Second File: support.c

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

```
 extern int count;
 void write_extern(void) {
   printf("count is %d\n", count);
}
```

Unions

- custom data type used for storing several variables in the same memory space
- access any of those variables at any time
- store / read from one of them at a time

Defining Unions

- define a union using the union keyword followed by the declarations of the union's members, enclosed in braces
- declare each member of a union as declaring a variable

```
union numbers {
      int i;
      float f;
};
```

Declaring Union Variables at Definition

```
union numbers {
      int i;
      float f;
} first_number, second_number;
```

Declaring Union Variables after Definition

```
union numbers {
      int i;
      float f;
};
union numbers first_number, second_number;
```

Initializing Union Members

```
union numbers {
      int         i;
      float       f; };
union numbers first_number = { 5 };
union numbers first_number = { f: 3.14159 };
union numbers first_number = { .f = 3.14159 };
```

Accessing Union Members

```
union numbers {
      int   i;
      float       f;
};

union numbers first_number;
```

```
first_number.i = 5; first_number.f = 3.9;
```

Union Practical Applications

- Online Payment
    - Credit Card
    - Debit Card
    - Net Banking
    - Wallet
- Dress Size
    - XXL
    - XL
    - L
    - M
    - S

Structures

A programmer-defined data type made up of variables of other data types

Defining Structures

define a structure using the struct keyword followed by the declarations of the structure's members, enclosed in braces.

```
struct point  {
    int x, y;
  };
```

Declaring Structure Variables at Definition

```
struct point  {
    int x, y;
  } first_point, second_point;
```

Declaring Structure Variables after Definition

```
struct point  {
    int x, y;
  };
struct point first_point, second_point;
```

Initializing Structure Members

```
struct point  {
    int x, y;
  };
struct point first_point = { 5, 10 };

struct point first_point = { .y = 10, .x = 5 };

struct point first_point = { y: 10, x: 5 };

struct point  {
    int x, y;
  } first_point = { 5, 10 };
```

```
struct pointy  {
    int x, y;
    char *p;
  };
struct pointy first_pointy = { 5 };
```

Here, `x` is initialized with 5, `y` is initialized with 0

```
struct point  {
    int x, y;
  };

struct rectangle  {
    struct point top_left, bottom_right;
  };

struct rectangle my_rectangle = { {0, 5}, {10, 0} };
```

Accessing Structure Members

```
first_point.x = 0;
first_point.y = 5;

my_rectangle.top_left.x = 0;
my_rectangle.top_left.y = 5;

my_rectangle.bottom_right.x = 10;
my_rectangle.bottom_right.y = 0;
```

Unions	Structures
Common storage space for all members	Individual storage space for each members
Occupies lower memory space	Occupies higher memory space
Can access only one member of union at a time	Can access all members of structure at a time

Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
enum State {Working = 1, Failed = 0};
```

The keyword 'enum' is used to declare new enumeration types in C.

```
enum flag{constant1, constant2, constant3, ....... };
enum week{Mon, Tue, Wed};
enum day;
```

// **Or**

```
enum week{Mon, Tue, Wed}day;
```

```
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
}
```

Output:

```
2
```

```
enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,Aug, Sep, Oct, Nov,
Dec};
int main()
{
   int i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
          wednesday, thursday = 10, friday, saturday};

int main()
{
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
           wednesday, thursday, friday, saturday);
}
```

Output: `1 2 5 6 10 11 12`

Advantages of using enum over macro: Enum variables are automatically assigned values.

Arrays

An array is an identifier that refers to a collection of data items of that have the same name. They must also have the same data type (i.e. all characters, all integers etc.). Each data item is represented by its array element. The individual array elements are distinguished from one another by their subscripts.

Syntax for array declaration:

Suppose arr is a 5 element integer array which stores the numbers 5, 4, 2, 7, 3 in that order. The first element is referred to as arr[0]which stores the data value 5, the second element is arr [1] which stores the value 4 and so on. The last element is arr [4] which stores the value 3. The subscript associated with each element is shown in square braces. For an n-element array the subscripts will range from 0 to n-1.

In case of a character array of size n, the array will be able to store only n-1 elements as a null character is automatically stored at the end of the string to terminate it. Therefore, a character array letter that stores 5 elements must be declared of size 6. The fifth element would be stored at letter [4] and letter [5] will store the null character.

Can store one or more elements consecutively in memory. Array elements are indexed beginning at position zero, not one.

The number of elements in an array must be positive.

Declare an array by specifying the data type for its elements, its name, and the number of elements it can store

```
int my_array[10];
```

The number of elements can be as small as zero.

Zero-length arrays are useful as the last element of a structure which is really a header for a variable-length object

```
struct line{
  int length;
  char contents[0];
};

{
  struct line *this_line = (struct line *)
    malloc (sizeof (struct line) + this_length);
  this_line -> length = this_length;
}
```

Array size can be variables

```
int my_function (int number){
  int my_array[number];
}
```

Initializing Arrays

Array elements can be initialized while declaring it by listing the initializing values, separated by commas, in a set of braces.

```
void main(void){
int i, my_array[5] = { 0, 1, 2, 3, 4 };
for(i=0;i<5;i++)
      printf("%d\n",my_array[i]);
}
```

Don't have to explicitly initialize all of the array elements.

For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
void main(void)
{
int i, my_array[5] = { 0, 1, 2 };
for(i=0;i<5;i++)
      printf("%d\n",my_array[i]);
}
```

Array elements can be initialized by specifying array indices by including the array index in brackets, and optionally the assignment operator, before the value.

```
int my_array[5] = { [2] 5, [4] 9 };
```

or,

```
int my_array[5] = { [2] = 5, [4] = 9 };
```

Both of those examples are equivalent to:

```
int my_array[5] = { 0, 0, 5, 0, 9 };
```

can initialize a range of elements to the same value, by specifying the first and last indices, in the form [*first*] ... [*last*] .

```
int my_array[10] = { [0 ... 3] = 1, [4 ... 8] = 2, 3 };
```

That initializes elements 0 through 3 to 1, elements 4 through 8 to 2, and element 9 to 3. (also could explicitly write [9] = 3.) Notice that *must* have spaces on both sides of the '...'

If every element of an array is initialized, then its size is determined by the number of elements initialized.

```
int my_array[] = { 0, 1, 2, 3, 4 };
```

If specified which elements to be initialized, then the size of the array is equal to the highest element number initialized, plus one.

```
int my_array[] = { 0, 1, 2, [9] = 9 };
```

In that example, only four elements are initialized, but the last one initialized is element number 9, so there are 10 elements.

Accessing Array Elements

Specifying the array name, followed by the element index, enclosed in brackets.

```
my_array[0] = 5;
```

That assigns the value 5 to the first element in the array, at position zero.

Multidimensional Arrays

Arrays of Arrays - A two-dimensional array that holds five elements in each dimension

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

two_dimensions[1][3] = 12;

# include <stdio.h>

void main(void){
int i,j, two_dimensions[2][5]= { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10} };
for(i=0;i<2;i++)
      for (j=0;j<5;j++)
            printf("%d\n",two_dimensions[i][j]);
}
```

Arrays as Strings

An array of characters to hold a string

```
char blue[26];

char yellow[26] = {'y', 'e', 'l', 'l', 'o', 'w', '\0'};

char orange[26] = "orange";

char gray[] = {'g', 'r', 'a', 'y', '\0'};

char salmon[] = "salmon";
```

The string terminator null character \0 is included at the end of the string

```
char lemon[26] = "custard";
lemon = "steak sauce";      /* Fails! */
```

Can change one character at a time

```
char name[] = "bob";

name[0] = 'r';
```

It is possible to initialize an array using a string that has more characters than the specified size.
This is not a good thing. The larger string will *not* override the previously specified size of the array, and you will get a compile-time warning.

Since the original array size remains, any part of the string that exceeds that original size is being written to a memory location that was not allocated for it.

```
char greet[5]=”Good Morning”;
```

Arrays of Unions

Array of a union type is created as an array of a primitive data type.

```
union numbers  {
    int i;
    float f;
  };
union numbers number_array [3];
```

Created a 3-element array of `union numbers` variables called `number_array`.

Initialize the first members of the elements of a number array:

```
union numbers number_array [3] = { {3}, {4}, {5} };
```

The additional inner grouping braces are optional.

After initialization, the union members are accessed in the array using the member access operator.

```
union numbers number_array [3];

number_array[0].i = 2;
```

Arrays of Structures

Array of a structure type is created as an array of a primitive data type.

```
struct point  {
    int x, y;
  };
struct point point_array [3];
```

Created a 3-element array of `struct point` variables called `point_array`.

Initialize the elements of a structure array:

```
struct point point_array [3] = { {2, 3}, {4, 5}, {6, 7} };
```

The additional braces are used for partially initializing some of the structures in the array, and fully initialize others:

```
struct point point_array [3] = { {2}, {4, 5}, {6, 7} };
```

The value 4 is assigned to the x member of the second array element, *not* to the y member of the first element, as would be the case without the grouping braces.

Access the structure members in the array using the member access operator.

```
struct point point_array [3];
```

```
point_array[0].x = 2;

point_array[0].y = 3;
```

Typedef

to give a type, a new name

```
typedef unsigned char BYTE;
```

the identifier BYTE can be used as an abbreviation for the type unsigned char

```
BYTE  b1, b2;
```

Preprocessors

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. A C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

Examples

```
#define MAX_ARRAY_LENGTH 20
#include <stdio.h>
#include "myheader.h"
#undef  FILE_SIZE
#define FILE_SIZE 42
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

Directive	**Description**
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.

#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

Preprocessor Operators

The Macro Continuation (\) Operator

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": Festival Greetings!\n")
```

The Stringize (#) Operator

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": Festival Greetings!\n")

int main(void) {
   message_for(Carole, Debra);
   return 0;
}
Carole and Debra: Festival Greetings!
```

The Token Pasting (##) Operator

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
   int token34 = 40;
   tokenpaster(34);
   return 0;
}
token34 = 40
```

The Defined() Operator

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void) {
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
Here is the message: You wish!
```

Parameterized Macros

```
#define square(x) ((x) * (x))
```

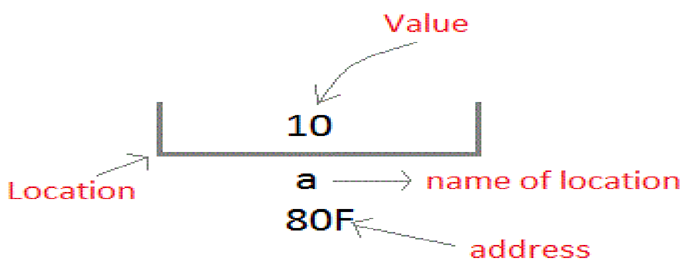
Predefined Macros

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

```
#include <stdio.h>

main() {

   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Pointers

A pointer is a variable whose value is the address of another variable. A pointer can be any variable type. The *unary* or *monadic* operator & gives the ``address of a variable''. The *indirection* or dereference operator * gives the ``contents of an object *pointed to* by a pointer''.



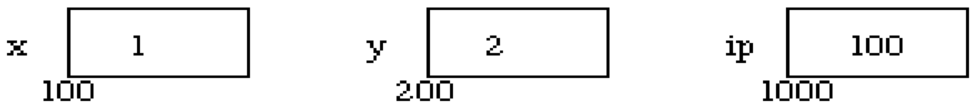
Declaring Pointers

```
data-type * name;
```

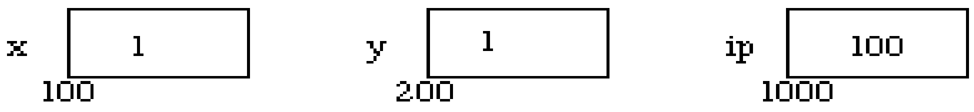
White space is not significant around the indirection operator:

```
data-type *name;

data-type* name;

char *p1; int *p2; float *p3; double *p4;
```

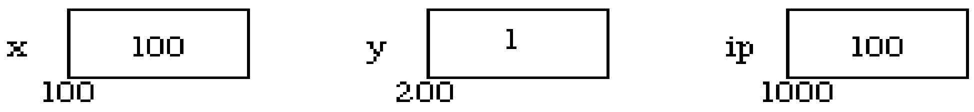
```
int x = 1, y =2;
int *ip;

ip = &x;
```



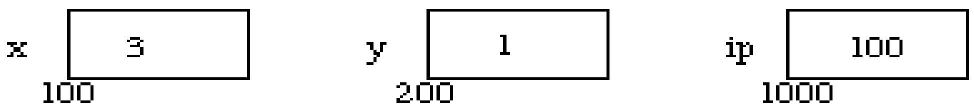
```
y = *ip;
```



```
x = ip;
```



```
*ip = 3
```



Initializing Pointers

```
int i;

int *ip = &i;

# include <stdio.h>
```

```
void main(void){
      int i=3, j=4, *ip, *jp;
ip = &i;
jp = &j;
printf("%u\t%u\t%u\t%u",i,j,ip,jp);
printf("%u\t%u\t%u\t%u",i,j,*ip,*jp);
}
```

```
int i, j;
int *ip = &i;   /* 'ip' now holds the address of 'i'.  */
ip = &j;        /* 'ip' now holds the address of 'j'.  */
*ip = &i;       /* 'j' now holds the address of 'i'.  */
```

Major advantages of pointers are:

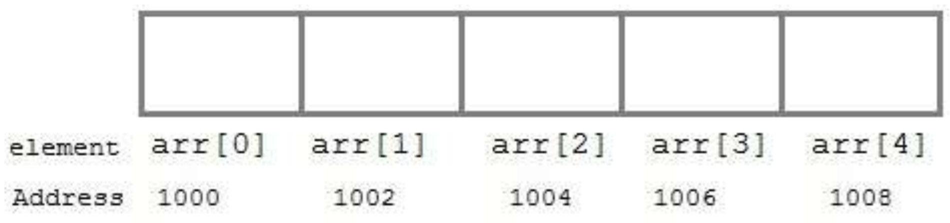
(i) It allows management of structures which are allocated memory dynamically.
(ii) It allows passing of arrays and strings to functions more efficiently.
(iii) It makes possible to pass address of structure instead of entire structure to the functions.
(iv) It makes possible to return more than one value from the function.

Arrays are closely related to pointers

```
#include <stdio.h>
int main(){
   char charArr[4];
   int i;

   for(i = 0; i < 4; ++i)
   {
      printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
   }

   return 0;
}
```

```
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47
```

int arr[4];
&arr[0] is equivalent to arr
arr[0] is equivalent to *arr
&arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).
&arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).
&arr[3] is equivalent to (arr + 3) AND, arr[3] is equivalent to *(arr + 3).
&arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).



Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

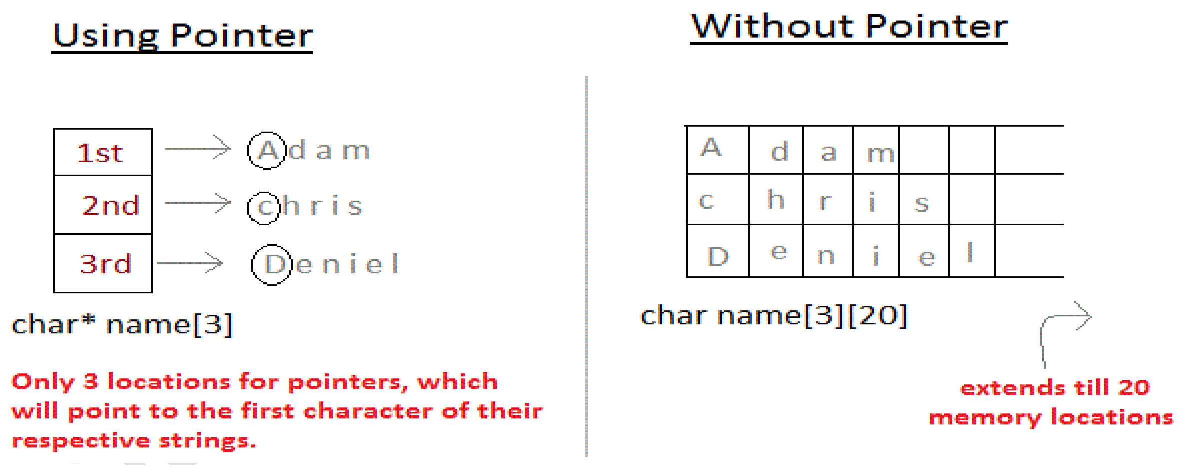
printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

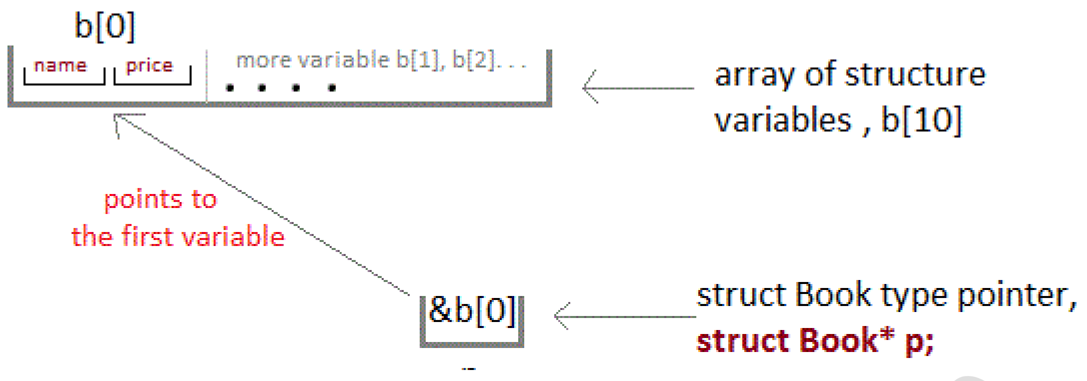
printf("%d", *a);⟶ **will print value of a[0] only**

a++;⟶ **Compile time error, we cannot change base address of the array.**


Using Pointer
Without Pointer
1st
2nd
3rd
Adam
Chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

A parameter is **passed by reference**, the caller and the callee **use the same variable** for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

A parameter is **passed by value**, the caller and callee have **two independent variables** with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller.


b[0]
name
price
more variable b[1], b[2]. . .
array of structure variables , b[10]
points to the first variable
&b[0]
struct Book type pointer,
struct Book* p;

Pointers to Unions

A pointer to a union type is created as a pointer to a primitive data type.

```
union numbers  {
    int i;
    float f;
  };
union numbers foo = {4};
union numbers *number_ptr = &foo;
```

Access the members of a union variable through a pointer

```
number_ptr -> i = 450;
```

Pointers to Structures

A pointer to a structure type is created as a pointer to a primitive data type.

```
struct fish  {
    float length, weight;
  };
struct fish salmon = {4.3, 5.8};
struct fish *fish_ptr = &salmon;
```

Access the members of a structure variable through a pointer

```
fish_ptr -> length = 5.1;

fish_ptr -> weight = 6.2;
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

```
int  *ptr = NULL;
```

To check for a null pointer, you can use an 'if' statement as follows

```
if(ptr)      /* succeeds if p is not null */
if(!ptr)     /* succeeds if p is null */
```

There are four arithmetic operators that can be used in pointers: ++, --, +, -

```
ptr++;
ptr--;
ptr=ptr+1;
ptr=ptr-1;
```

Memory allocation – Dynamic – Run time

malloc() allocates requested size of bytes and returns a pointer first byte of allocated space

ptr = (cast-type*) malloc(byte-size)

```
ptr = (int*) malloc(100 * sizeof(int));
```

calloc() allocates space for an array elements, initializes to zero and then returns a pointer

ptr = (cast-type*) calloc(n, element-size);

```
ptr = (float*) calloc(25, sizeof(float));
```

free() Deallocate the previously allocated space

```
free(ptr);
```

realloc() Change the size of previously allocated space
ptr = realloc(ptr, newsize);

```
ptr = realloc(ptr, 40);
```

Program to swap two number using call by reference.

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main(){
    int num1 = 5, num2 = 10;

    swap( &num1, &num2);
    printf("Number1 = %d\n", num1);
    printf("Number2 = %d", num2);
    return 0;
}

void swap(int * n1, int * n2){
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

```
#include <stdio.h>
#include <conio.h>
int* larger(int*, int*);
void main(){
 int a=15;
 int b=92;
 int *p;
 p=larger(&a, &b);
 printf("%d is larger",*p);
}
int* larger(int *x, int *y){
 if(*x > *y)
  return x;
 else
  return y;
}
```

Multiple indirection

```
int    a =  3;
int   *b = &a;
int  **c = &b;
int ***d = &c;

***d == **c == *b == a == 3;
```

Constant Pointers

These two declarations are equivalent

```
const int *ptr_a;

int const *ptr_b;
```

These two declarations are not equivalent

```
int const *ptr_c;
```

you cannot change `*ptr_c = 42;` you can `*ptr_c++;`

```
int *const ptr_d;
```

you can change `*ptr_d = 42;` you cannot `ptr_d++;`

Benefits(use) of pointers in c:

1. Pointers provide direct access to memory
2. Pointers provide a way to return more than one value to the functions
3. Reduces the storage space and complexity of the program
4. Reduces the execution time of the program
5. Provides an alternate way to access array elements
6. Pointers can be used to pass information back and forth between the calling function and called function.
7. Pointers allow us to perform dynamic memory allocation and deallocation.

8. Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
9. Pointers allow us to resize the dynamically allocated memory block.
10. Addresses of objects can be extracted using pointers

Drawbacks of pointers in c:

1) Uninitialized pointers might cause segmentation fault.
2) Dynamically allocated block needs to be freed explicitly.Otherwise, lead to memory leak.
3) If pointers are updated with incorrect values, it might lead to memory corruption.
4) Pointer bugs are difficult to debug.

Data Files

File is a place on your Disk (secondary memory-permanent) where information is stored.

Storing in a file will preserve your data even if the program terminates.

One time entry of large number of data

Moving data from one computer to another

Types of Files

There are two types of files.

1. Text files

Text files are the normal .txt files that can be easily created using Notepad or any text editors. The contents of the file is plain text and visible. Edit or delete the contents easily. They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer. Instead of storing data in plain text, they store it in the binary form (0's and 1's). They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, there are six major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading information from a file
5. Writing information to a file
6. Seeking information in a file

Working with files

Declare a pointer of type file; this declaration is needed for communication between the file and program.

```
FILE *fptr;
```

Opening a file - for creation and edit

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode")
```

For Example:

```
fopen("E:\\cprogram\\test.dat","w");
fopen("E:\\cprogram\\test.dat","rb");
```

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'. The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'. The reading mode only allows you to read the file, you cannot write into the file.

Closing a File

The file (both text and binary) should be closed after reading/writing. Closing a file is performed using library function fclose().

```
fclose(fptr);
```

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

Write to a text file using fprintf()

```
#include <stdio.h>
#include <stdlib.h>
void main(){
      int num;
      FILE *fptr;
      fptr = fopen("test.dat","w");
      if(fptr == NULL)         {
             printf("Error!");
             exit(1); // Program exits if the file pointer returns NULL.
       }
      printf("Enter num: ");
      scanf("%d",&num);
       fprintf(fptr,"%d",num);
       fclose(fptr);
}
```

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append.	If the file does not exists, it will be created.
ab	Open for append in binary mode.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Read from a text file using fscanf()

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int num;
   FILE *fptr;

   if ((fptr = fopen("test.dat","r")) == NULL){
       printf("Error! opening file");
       exit(1); // Program exits if the file pointer returns NULL.
   }
   fscanf(fptr,"%d",&num);

   printf("Value of n=%d",num);
   fclose(fptr);

   return 0;
}
```

Writing to a binary file using fwrite()

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

struct date{
   int n1, n2, n3;
};
int main(){
   int n;
   struct date num;
   FILE *fptr;
   if ((fptr = fopen("test.dat","wb")) == NULL){
       printf("Error! opening file");
       exit(1); // Program exits if the file pointer returns NULL.
   }
   for(n = 1; n < 5; ++n)   {
      num.n1 = n;
      num.n2 = 5*n;
      num.n3 = 5*n + 1;
      fwrite(&num, sizeof(struct date), 1, fptr);
   }
   fclose(fptr);
   return 0;
}
```

REMOVING a FILE

```
remove("test.dat");
```

Reading from a binary file using fread()

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

struct date{
   int n1, n2, n3;
};

int main(){
   int n;
   struct date num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");
       exit(1); // Program exits if the file pointer returns NULL.
   }
   for(n = 1; n < 5; ++n)   {
      fread(&num, sizeof(struct date), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
   }
   fclose(fptr);
   return 0;
}
```

Seeking the cursor to the given record in the file

```
fseek (FILE * stream, long int offset, int whence)
```

Whence	**Meaning**
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

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

struct date{
   int n1, n2, n3;
};

int main(){
   int n;
   struct date num;
   FILE *fptr;
   if ((fptr = fopen("test.dat","rb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   // Moves the cursor to the end of the file
   fseek(fptr, sizeof(struct date), SEEK_END);
   for(n = 1; n < 5; ++n)   {
      fread(&num, sizeof(struct date), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);
   return 0;
}
```

ftell() - It tells the byte location of current position of cursor in file pointer.
rewind() - It moves the control to beginning of the file.

```
printf("n1: %d\tn2: %d\tn3: %d\n%d\t", num.n1, num.n2, num.n3,ftell(fptr));

rewind (fptr);
```

EOF - end of file

```
if (feof(fp))
     printf("\n End of file reached.");
#include<stdio.h>
```

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

void main(void) {
   FILE *fp1, *fp2;
   char ch;
   clrscr();
   fp1 = fopen("Sample.txt", "r");
   fp2 = fopen("Output.txt", "w");
   while (1) {
      ch = fgetc(fp1);
      if (ch == EOF) break;
      else putc(ch, fp2);
   }
   printf("File copied Successfully!");
   fclose(fp1);
   fclose(fp2);
}
```

r+ does not truncate (delete) the content of file as well it doesn't create a new file if such file doesn't exits while in w+ truncate the content of file as well as create a new file if such file doesn't exists.

```
int getc(FILE *fp)
int putc(int char, FILE *fp)
int fgetc (FILE *fp);
int fputc(int c, FILE *fp );
int fgets(char *buf, int n, FILE *fp);   reads string from a file, one line at a time.
int fputs(const char *s, FILE *fptr);  writes string to a file.
int getw (FILE *fp)  function reads an integer from file
int putw (int no, FILE *fp)  functions writes an integer to file
```

Binary file I/O - fread and fwrite

Both of these functions deal with blocks of memories - usually arrays / pointers / structures / unions

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE
*a_file);
size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements,
FILE *a_file);
```

Example

```
FILE *fp;
fp=fopen("c:\\test.bin", "wb");
char x[10]="ABCDEFGHIJ";
fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), fp);
```

A **system call** is just what its name implies—a request for the operating system to do something on behalf of the user's program   #include <fcntl.h>

1. open()
2. read()
3. write()
4. close()

1. open()

open is rather like the fopen, except that instead of returning a file pointer, it returns a file descriptor, which is just an int. open returns -1 if any error occurs.

```
int open( char *filename, int access, int permission );
```

The available access modes are `O_RDONLY  O_WRONLY  O_RDWR  O_APPEND  O_BINARY  O_TEXT`
The permissions are `S_IWRITE  S_IREAD  S_IWRITE | S_IREAD`

2. read()

```
int read( int handle, void *buffer, int nbyte );
```

The read() function attempts to read nbytes from the file associated with handle, and places the characters read into buffer. If the file is opened using O_TEXT, it removes carriage returns and detects the end of the file. The function returns the number of bytes read. On end-of-file, 0 is returned, on error it returns -1, setting errno to indicate the type of error that occurred.

3. write()

```
int write( int handle, void *buffer, int nbyte );
```

The write() function attempts to write nbytes from buffer to the file associated with handle. On text files, it expands each LF to a CR/LF. The function returns the number of bytes written to the file. A return value of -1 indicates an error, with errno set appropriately.

4. close()

```
int close( int handle );
```

The close() function closes the file associated with handle. The function returns 0 if successful, -1 to indicate an error, with errno set appropriately.

Implementation of open(), read(), write() and close() functions

```
#include <stdio.h>
#include <fcntl.h>
int main(){
int fd;
char buffer[80];
static char message[]="Hello, SPCE - Visnagar";
fd=open("myfile.txt",O_RDWR);
if (fd != -1){
      printf("myfile.txt opened with read/write access\n");
      write(fd,message,sizeof(message));
      lseek(fd,0,0);
      read(fd,buffer,sizeof(message));
      printf("%s — was written to myfile.txt \n",buffer);
      close(fd);
}
}
```

FILE *fp;

fp=fopen (“filename”, ”‘mode”);

int fclose(FILE *fp);

int feof(FILE *fp); // returns a non-zero value when End-of-File else zero is returned.

int fgetc(FILE *fp);

int fputc(int char, FILE *fp);

int getc(FILE *fp);

int putc(int char, FILE *fp);

int getch(void); // it won’t echo the input character on to the output screen

int getche(void);

int fputs (const char *string, FILE *fp);

char *fgets(char *string, int n, FILE *fp)

int putw(int number, FILE *fp);

int getw(FILE *fp);

int fscanf(FILE *fp, const char *format, variable list);

int fprintf(FILE *fp, const char *format, variable list);

int fseek(FILE *fp, long int offset, int whence); offset – Number of bytes to be moved from whence

long int ftell(FILE *fp);

void rewind(FILE *fp);

int remove(const char *filename);

int fgetchar(void); //only from keyboard

int fputchar(int c); //only to display device

int sprintf(char *string, const char *format, variable list); // sprintf (string, “%d %c %f”, value, c, flt) ;

int sscanf(const char *string, const char *format, variable list); // sscanf (buffer,”%s
%d”,name,&age);

‘C’ Function Exercises:

1. Write a C program to find cube of any number using function.
2. Write a C program to find diameter, circumference and area of circle using functions. (input is radius)
3. Write a C program to find maximum and minimum between two numbers using functions.
4. Write a C program to check whether a number is even or odd using functions.
5. Write a C program to find all prime numbers between given interval using functions.
6. Write a C program to print all strong numbers between given interval using functions. (Strong numbers are the numbers whose sum of factorial of digits is equal to the original number. Example: 145 is a strong number)
7. Write a C program to print all armstrong numbers between given interval using functions. (sum of the cubes of its digits is equal to the number itself)
8. Write a C program to print all perfect numbers between given interval using functions. (a perfect number is a positive integer equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself)
9. Write a C program to find power of any number using recursion.
10. Write a C program to print all natural numbers between 1 to n using recursion.
11. Write a C program to find sum of all natural numbers between 1 to n using recursion.
12. Write a C program to find reverse of any number using recursion.
13. Write a C program to check whether a number is palindrome or not using recursion.
14. Write a C program to find sum of digits of a given number using recursion.
15. Write a C program to find factorial of any number using recursion.
16. Write a C program to generate n^{th} Fibonacci term using recursion.
17. Write a C program to find Greatest Common Divisor (GCD) / Highest Common Factor (HCF) of two numbers using recursion.
18. Write a C program to find Least Common Multiple (LCM) of two numbers using recursion.
19. Write a C program to display all array elements using recursion.
20. Write a C program to find sum of elements of array using recursion.
21. Write a C program to find maximum and minimum elements in array using recursion.
22. Write functions to calculate the sine and cosine of their input. Choose appropriate types for both argument and return value. The series (given below) can be used to approximate the answer. The function should return when the value of the final term is less than 0.000001 of the current value of the function.
    sin x = x - pow(x,3)/fact(3) + pow(x,5)/fact(5)...
    cos x = 1 - pow(x,2)/fact(2) + pow(x,4)/fact(4)...

Note the fact that the sign in front of each term alternates (--+--+--+...). pow (x,n) returns x to the *n*th power, fact(n) factorial of n ($1 \times 2 \times 3 \times \cdots \times n$). You will have to write such functions. Check the results against published tables.

23. Write and test a recursive function that performs the admittedly dull task of printing a list of numbers from 100 down to 1. On entry to the function it increments a static variable. If the variable has a value below 100, it calls itself again. Then it prints the value of the variable, decrements it and returns. Check that it works.

‘C’ Arrays Exercises:

1. Write a program in C to read n number of values in an array and display it in reverse order.
2. Write a program in C to count a total number of duplicate / unique elements in an array.
3. Write a program in C to print all unique elements in an array.
4. Write a program in C to merge two arrays of same size sorted in ascending order.
5. Write a program in C to count the frequency of each element of an array.
6. Write a program in C to find the maximum and minimum element in an array.
7. Write a program in C to separate odd and even integers in separate arrays.
8. Write a program in C to sort elements of array in ascending / descending order.
9. Write a program in C to insert New value in the array (sorted list).
10. Write a program in C to delete an element at desired position from an array.
11. Write a program in C to find the second largest / smallest element in an array.
12. Write a program in C for addition / subtraction / multiplication / transpose of two Matrices of same size.
13. Write a program in C to calculate determinant of a 3 x 3 matrix.
14. Write a program in C to accept two matrices and check whether they are equal.
15. Write a program in C to check whether a given matrix is an identity matrix.

‘C’ Pointer Exercises:

1. Write a program in C to add two numbers using pointers.
2. Write a program in C to add numbers using call by reference.
3. Write a program in C to store n elements in an array and print the elements using pointer.
4. Write a program in C to print all permutations of a given string using pointers.
5. Write a program in C to calculate the length of the string using a pointer.
6. Write a program in C to swap two elements using call by reference.
7. Write a program in C to find the factorial of a given number using pointers.
8. Write a program in C to count the number of vowels and consonants in a string using a pointer.
9. Write a program in C to sort an array of numbers using Pointer.
10. Write a program in C to print a string in reverse using a pointer.

‘C’ File Exercises:

1. Write a program in C to read the file and store the lines into an array.
2. Write a program in C to Find the Number of Lines in a Text File.
3. Write a program in C to count a number of words and characters in a file.
4. Write a program in C to delete a specific line from a file.
5. Write a program in C to replace a specific word with another word in a file.
6. Write a program in C to merge two files and write it in a new file.
7. Write a program in C to encrypt and decrypt a text file.

Program samples

```
#include <stdio.h>  //swap two numbers without a temporary variable
void main(){
  int x = 10, y = 5;
  x = x + y;  // x now becomes 15   // using + and -
  y = x - y;  // y becomes 10
  x = x - y;  // x becomes 5
      x = x * y;  // x now becomes 50           // using * and /
      y = x / y;  // y becomes 10
      x = x / y;  // x becomes 5
  x = x ^ y;  // x now becomes 15 (1111) //using ^XOR logical addition
  y = x ^ y;  // y becomes 10 (1010)
  x = x ^ y;  // x becomes 5 (0101)
  printf("After Swapping: x = %d, y = %d", x, y);
}
#include <stdio.h> // print "Hello C" using if-else statement both?
void main(){
   if(!printf("Hello ")) ;
   else  printf("C\n");
}
#include <stdio.h> // find odd or even using bit wise operator
void main(){
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    if(x&1)      printf("%d is ODD\n", x);
    else         printf("%d is EVEN\n", x);
}
Number is odd or even without any relational and arithmetic operators?
#include <stdio.h>
void main(){
    int number;
    printf("Please input an integer number: ");
    scanf("%d",&number);
    (number & 0x01) ? puts("odd") :  puts ("even");
    printf("\n");
}
#include <stdio.h> // Given number is positive or negative or zero.
void main(){
    float n;
    printf("Please enter a number\n");
    scanf("%f",&n);
    if (n==0)printf("\nEntered number is 0. It's neither positive or
negative");
    else if (n<0)  printf("\nEntered number %.2f is negative.",n);
    else printf("\nEntered number %.2f is positive.",n);
}
#include <stdio.h> // factorial
void main(){
    int n, i;
    unsigned long long factorial = 1;
    scanf("%d",&n);
    for(i=1; i<=n; ++i) factorial *= i;     // factorial = factorial*i;
    printf("Factorial of %d = %llu", n, factorial);
}
#include<stdio.h>     //Fibonacci
void main(){
   int n, first = 0, second = 1, next, c;
   scanf("%d",&n);
   for ( c = 0 ; c < n ; c++ ){
      if ( c <= 1 ) next = c;
```

```
       else{
          next = first + second;
          first = second;
          second = next;
       }
       printf("%d\n",next);
    }
}
#include <stdio.h>             // LCM
void main(){
    int n1, n2, LCM;
    scanf("%d %d", &n1, &n2);
    LCM = (n1>n2) ? n1 : n2;
    while(1){
        if(LCM %n1==0 && LCM %n2==0 )         {
            printf("The LCM of %d and %d is %d.",n1, n2, LCM);
            break;
        }
        ++ LCM;
    }
}
#include <stdio.h>            // GCD HCF
void main(){
    int n1, n2, i, gcd;
    scanf("%d %d", &n1, &n2);
    for(i=1; i <= n1 && i <= n2; ++i)
        if(n1%i==0 && n2%i==0) gcd = i;
            printf("G.C.D of %d and %d is %d", n1, n2, gcd);
}
#include <stdio.h>            // gcd hcf using recursion
int hcf(int n1, int n2);
void main(){
   int n1, n2;
   printf("Enter two positive integers: ");
   scanf("%d %d", &n1, &n2);
   printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
}
int hcf(int n1, int n2){
    if (n2 != 0) return hcf(n2, n1%n2);
    else return n1;
}
#include <stdio.h>   // calculate the number of digits
void main(){
    long long n;   int count = 0;
    printf("Enter an integer: ");
    scanf("%lld", &n);
    while(n != 0){
        n /= 10;   ++count;
    }
    printf("Number of digits: %d", count);
}
#include <stdio.h>  // palindrome number
void main(){
    int n, reversedInteger = 0, remainder, originalInteger;
    printf("Enter an integer: ");
    scanf("%d", &n);
    originalInteger = n;
    while( n!=0 )    {
        remainder = n%10;
        reversedInteger = reversedInteger*10 + remainder;
        n /= 10;
    }
    if (originalInteger == reversedInteger)
        printf("%d is a palindrome.", originalInteger);
    else
        printf("%d is not a palindrome.", originalInteger);
}
#include <stdio.h>     //palindrome string
#include <string.h>
void main(){
   char a[100], b[100];
   printf("Enter the string to check if it is a palindrome\n");
   gets(a);    strcpy(b,a);    strrev(b);
   if (strcmp(a,b) == 0)      printf("Entered string is a palindrome.\n");
   else     printf("Entered string is not a palindrome.\n");
}
# include <stdio.h>  // printing an array in reverse order
void main(void){
int n,i;
int no[n];
scanf("%d",&n);
for(i=0;i<n;i++)  scanf("%d",&no[i]);
for(i=n-1;i>=0;i--) printf("%d",no[i]);
}
#include <stdio.h> // sorting an array
void main(){
  int array[100], n, c, d, swap;
  scanf("%d", &n);
  for (c = 0; c < n; c++)    scanf("%d", &array[c]);
  for (c = 0 ; c < ( n - 1 ); c++)
    for (d = 0 ; d < n - c - 1; d++)
      if (array[d] < array[d+1]) /* For decreasing order use < */      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
  }
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )      printf("%d\n", array[c]);
}
#include<stdio.h> //finding duplicates
void main(void){
  int arr[] = {4, 2, 4, 5, 2, 3, 1}, i, j;
  int size = sizeof(arr)/sizeof(arr[0]);
  printf(" Repeating elements are ");
  for(i = 0; i < size; i++)
    for(j = i+1; j < size; j++)
      if(arr[i] == arr[j])printf(" %d ", arr[i]);
}
#include<stdio.h> //matrix addition, subtraction, multiplication
void main() {
   int i,j,c,r,k;
   int a[20][20],b[20][20],ma[20][20],ms[20][20];
   int mm[20][20];
   scanf("%d%d",&c,&r);
   for(i=0;i<c;i++)      {
       for(j=0;j<r;j++)  scanf("%d",&a[i][j]);
       printf("\n");
     }
   for(i=0;i<c;i++)      {
       for(j=0;j<r;j++)  scanf("%d",&b[i][j]);
       printf("\n");
     }
   for(i=0;i<c;i++)
       for(j=0;j<r;j++)         {
          ma[i][j]=a[i][j]+b[i][j];
          ms[i][j]=a[i][j]-b[i][j];
         }
   for(i=0;i<c;i++)
       for(j=0;j<r;j++)         {
           mm[i][j]=0;
           for(k=0;k<c;k++) mm[i][j] +=a[i][k]*b[k][j];
         }
   for(i=0;i<c;i++)     {
       for(j=0;j<r;j++)     printf("\t\t%d",ma[i][j]);
       printf("\n");
     }
   for(i=0;i<c;i++)     {
       for(j=0;j<r;j++)     printf("\t\t%d",ms[i][j]);
       printf("\n");
     }
   for(i=0;i<c;i++)     {
       for(j=0;j<r;j++)     printf("\t\t%d",mm[i][j]);
       printf("\n");
     }
 }
#include<stdio.h>       // sorting strings
#include <string.h>
void main(){
  int i,j,n;
  char str[20][20],temp[20];
  scanf("%d",&n);
  for(i=0;i<=n;i++)      gets(str[i]);
  for(i=0;i<=n;i++)
      for(j=i+1;j<=n;j++)
           if(strcmp(str[i],str[j])>0){
               strcpy(temp,str[i]);
              strcpy(str[i],str[j]);
              strcpy(str[j],temp);
           }
  printf("The sorted string\n");
  for(i=0;i<=n;i++) puts(str[i]);
}
#include <stdio.h>      // find number of words in given string
#include <string.h>
void main(){
    char s[200]; int count = 0, i;
    scanf("%[^\n]s", s);
    for (i = 0;s[i] != '\0';i++)
        if (s[i] == ' ') count++;
          printf("number of words in given string are: %d\n", count + 1);
}
#include<stdio.h> // delete duplicate values from an array.
void main() {
   int arr[20], i, j, k, size;
   printf("\n Enter array size : ");    scanf("%d", &size);
   printf("\n Accept Numbers : ");
   for (i = 0; i < size; i++)       scanf("%d", &arr[i]);
   printf("\n Array with Unique list  : ");
   for (i = 0; i < size; i++) {
      for (j = i + 1; j < size;) {
         if (arr[j] == arr[i]) {
            for (k = j; k < size; k++)
               arr[k] = arr[k + 1];
```

```
             size--;
          } else   j++;      }    }
    for (i = 0; i < size; i++)
      printf("%d ", arr[i]);
}
```
Program to print half pyramid a using numbers
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```
```
#include <stdio.h>
void main(){
    int i, j, rows;
    printf("Enter number of rows: "); scanf("%d",&rows);
    for(i=1; i<=rows; ++i){
        for(j=1; j<=i; ++j) printf("%d ",j);
        printf("\n");
    }
}
```
Inverted half pyramid using numbers
```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```
```
#include <stdio.h>
void main(){
    int i, j, rows;
    scanf("%d",&rows);
    for(i=rows; i>=1; --i){
        for(j=1; j<=i; ++j)
            printf("%d ",j);
        printf("\n");
    }
}
```
Program to print half pyramid a using alphabets
```
A
B B
C C C
D D D D
E E E E E
```
```
#include <stdio.h>
void main(){
    int i, j;
    char input, alphabet = 'A';
    printf("Enter the uppercase character you want to print in last row:
");
    scanf("%c",&input);
    for(i=1; i <= (input-'A'+1); ++i){
        for(j=1;j<=i;++j)
            printf("%c", alphabet);
        ++alphabet;
        printf("\n");
    }
}
```
Program to print pyramid using numbers
```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
```

```
#include <stdio.h>
void main(){
    int i, space, rows, k=0, count = 0, count1 = 0;
    scanf("%d",&rows);
    for(i=1; i<=rows; ++i)     {
        for(space=1; space <= rows-i; ++space){
            printf("  ");
            ++count;
        }
        while(k != 2*i-1){
            if (count <= rows-1){
                printf("%d ", i+k);
                ++count;
            }
            else{
                ++count1;
                printf("%d ", (i+k-2*count1));
            }
            ++k;
        }
        count1 = count = k = 0;
        printf("\n");
    }
}
```

Write a C-program to print the following pattern.

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

```
#include <stdio.h>  // star pyramid
void main(){
    int i, space, rows, k=0, middle=20;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=1; i<=rows; ++i, k=0)    {
        for(space=1; space<=middle-i; ++space)   printf("  ");
        while(k != 2*i-1)        {
            printf("* ");  // printf("%d ",k);
            ++k;
        }
        printf("\n");
    }
  }
```

Program to print pyramid using numbers

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

```
#include <stdio.h>
void main(){
    int i, space, rows, k=0, count = 0, count1 = 0;
    scanf("%d",&rows);
    for(i=1; i<=rows; ++i){
        for(space=1; space <= rows-i; ++space)        {
            printf("  ");
            ++count;
        }
        while(k != 2*i-1){
            if (count <= rows-1){
                printf("%d ", i+k);
                ++count;
            }
            else{
                ++count1;
                printf("%d ", (i+k-2*count1));
            }
            ++k;
        }
        count1 = count = k = 0;
        printf("\n");
    } }
#include <stdio.h>    // Right align half pyramid
void main(){
    int i, j, k, l, r=24, c=80, p=c/2, v=1, t=5;      //            1
    printf("Enter number of rows: ");           //      2       1
    scanf("%d",&r);                       //    3      2       1
    for(i=1; i<=r; ++i)    {
        for(j=0; j <= p; j++) printf(" ");
        for (k=1;k<=i;k++)    printf(" %d  ", k);
      printf("\n");
      p-=4;
} }
```

A number is considered as perfect number when it satisfies the below conditions. It should be a positive number When the sum of it's divisors (excluding that number) are equal to that number Example for perfect numbers: 6, 28, 496, 8128

Description:

1 + 2 + 3 = 6

1 + 2 + 4 + 7 + 14 = 28

```
 #include<stdio.h>
void main(){
     int number, sum=0, i=1;
     printf("Please enter a number to check perfect number\n");
     scanf("%d",&number);
      while(i<number)     {
           if(number%i==0)
               sum=sum+i;
           i++;
     }
      if(sum==number)
           printf("\nEntered number %d is a perfect number",i);
     else
          printf("\nEntered number %d is not a perfect number",i);
}
```

Pascal Triangle pattern:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

```
#include <stdio.h>        //factorial using recursion
long int multiplyNumbers(int n);
void main(){
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
}
long int multiplyNumbers(int n){
    if (n >= 1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
# include <stdio.h> // matrix multiplication
void main() {
  int a[10][10], b[10][10], m[10][10], r1, c1, r2, c2, i, j, k;
    scanf("%d %d", &r1, &c1);
    scanf("%d %d",&r2, &c2);
    for(i=0; i<r1; ++i)
        for(j=0; j<c1; ++j)
            scanf("%d", &a[i][j]);
    for(i=0; i<r2; ++i)
        for(j=0; j<c2; ++j)
            scanf("%d",&b[i][j]);
    for(i=0; i<r1; ++i)
        for(j=0; j<c2; ++j)
            for(k=0; k<c1; ++k)
                m[i][j]+=a[i][k]*b[k][j];
    for(i=0; i<r1; ++i)
        for(j=0; j<c2; ++j)
            printf("%d  ", m[i][j]);
        printf("\n\n");
}
# include <stdio.h> // switch example
void main() {
    char operator;
    double firstNumber,secondNumber;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%lf %lf",&firstNumber, &secondNumber);
    switch(operator)
    {
        case '+': printf("%.1lf + %.1lf = %.1lf",firstNumber, secondNumber, 
firstNumber+secondNumber); break;
        case '-': printf("%.1lf - %.1lf = %.1lf",firstNumber, secondNumber, 
firstNumber-secondNumber); break;
        case '*': printf("%.1lf * %.1lf = %.1lf",firstNumber, secondNumber, 
firstNumber*secondNumber); break;
        case '/': printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, 
firstNumber/firstNumber); break;
        default: printf("Error! operator is not correct");
    }
}
# include <stdio.h> // enum example
enum suit {
     club = 0,
     diamonds = 10,
     hearts = 20,
     spades = 3,
     ace = 25,
     joker = 50
} card;
void main()
```

```
{
      card = club;
      printf("Size of enum variable = %d bytes", sizeof(card));
}
#include<stdio.h>       //Fibonacci series using recursion
void printFibonacci(int);
int main(){
    int k,n;
    long int i=0,j=1,f;
    scanf("%d",&n);
    printf("%d %d ",0,1);
    printFibonacci(n-2);
}
void printFibonacci(int n){
    static long int first=0,second=1,sum;
    if(n>0){
         sum = first + second;
         first = second;
         second = sum;
         printf("%ld ",sum);
         printFibonacci(n-1);     } }
#include <stdio.h> // lcm using recursion
int lcm(int, int);
void main(){
    int a, b, result;
    int prime[100];
    printf("Enter two numbers: ");
    scanf("%d%d", &a, &b);
    result = lcm(a, b);
    printf("The LCM of %d and %d is %d\n", a, b, result);
}
int lcm(int a, int b){
    static int common = 1;
    if (common % a == 0 && common % b == 0) return common;
    common++;
    lcm(a, b);
}
#include <stdio.h>  // sorting strings using pointers
void main(){
char *str[10], *t; int i,j,n;
scanf("%d",&n);
for(i=0;i<n;i++)      scanf("%s",&str[i]);
for(i=0; i<n; i++)
    for(j=i+1; j<n; j++)
        if (strcmp(&str[j-1], &str[j]) > 0){
            t=str[j]; str[j]=str[j-1]; str[j-1]=t;
        }
printf("\n");
for(i=0;i<5;i++) printf("%s\n",&str[i]);
}
#include <stdio.h> // to add two complex numbers:
struct complex {
     int real, img;
};
void main(){
     struct complex a, b, c;
     printf("Please enter first complex number\n");
     printf("Enter Real part of the 1st complex number\n");
     scanf("%d", &a.real);
     printf("Enter Imaginary part of the 1st complex number without i\n");
     scanf("%d", &a.img);
     printf("Please enter second complex number\n");
     printf("Enter Real part of the 2nd complex number\n");
     scanf("%d", &b.real);
     printf("Enter Imaginary part of the 2nd complex number without i\n");
     scanf("%d", &b.img);
     c.real = a.real + b.real;
     c.img = a.img + b.img;
      if ( c.img >= 0 )          printf("The sum of two complex numbers =
%d + %di\n",c.real,c.img);
     else           printf("The sum of two complex numbers = %d
%di\n",c.real,c.img);
}
#include <stdio.h>      //swap two numbers with pointer and function
void swap(int *xp, int *yp){
    if (xp == yp)  return; // Check if the two addresses are same
    *xp = *xp + *yp;
    *yp = *xp - *yp;
    *xp = *xp - *yp;
}
void main(){
  int x = 10, y=5;   swap(&x, &y);
  printf("After swap: x = %d, y=%d", x,y);
}
#include <stdio.h>        // incrementing a pointer
const int MAX = 3;
void main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   ptr = var;
   for ( i = 0; i < MAX; i++) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      ptr++;
   }
}
#include <stdio.h>       //decrementing a pointer
const int MAX = 3;
void main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   ptr = &var[MAX-1];
   for ( i = MAX; i > 0; i--) {
      printf("Address of var[%d] = %x\n", i-1, ptr );
      printf("Value of var[%d] = %d\n", i-1, *ptr );
      ptr--;
   }
}
#include <stdio.h>      // comparing pointers
const int MAX = 3;
void main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] ) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      ptr++;
      i++;
   }
}
```

```
#include <stdio.h>
void main () {
    struct Books {
           char  title[50];
           char  author[50];
           char  subject[100];
           int   book_id;
    } *i;
    printf("%d", sizeof(i));
}
#include<stdio.h>    //stack implementation using pointers
#include<conio.h>
#include<stdlib.h>
#define MAX 50
int size;
struct stack {
   int arr[MAX];
   int top;
};
void init_stk(struct stack *st) {
   st->top = -1;
}
void push(struct stack *st, int num) {
   if (st->top == size - 1) {
      printf("\nStack overflow(i.e., stack full).");
      return;
   }
   st->top++;
   st->arr[st->top] = num;
}
int pop(struct stack *st) {
   int num;
   if (st->top == -1) {
      printf("\nStack underflow(i.e., stack empty).");
      return 0;
   }
   num = st->arr[st->top];
   st->top--;
   return num;
}
void display(struct stack *st) {
   int i;
   for (i = st->top; i >= 0; i--)
      printf("\n%d", st->arr[i]);
}
void main() {
   int element, opt, val;
   struct stack ptr;
   init_stk(&ptr);
   printf("\nEnter Stack Size :");
   scanf("%d", &size);
   while (1) {
      printf("\n\ntSTACK PRIMITIVE OPERATIONS");
      printf("\n1.PUSH");
      printf("\n2.POP");
      printf("\n3.DISPLAY");
      printf("\n4.QUIT");
      printf("\n");
      printf("\nEnter your option : ");
      scanf("%d", &opt);
      switch (opt) {
```

```
      case 1:
         printf("\nEnter the element into stack:");
         scanf("%d", &val);
         push(&ptr, val);
         break;
      case 2:
         element = pop(&ptr);
         printf("\nThe element popped from stack is : %d", element);
         break;
      case 3:
         printf("\nThe current stack elements are:");
         display(&ptr);
         break;
      case 4:
         exit(0);
      default:
         printf("\nEnter correct option!Try again.");
      }
   }
}
#include<stdio.h> // length of string using pointer
void main() {
  char str[20], *pt;
  int i = 0;
  gets(str);
  pt = str;
  while (pt[i++]);
  printf("Length of String : %d", i-1);
}
#include<stdio.h> // string reverse using pointer
void main() {
  char str[20], *pt;
  int i = 0;
  gets(str);
  pt = str;
  while (pt[i++]);
  while (i>=0)
      putchar(pt[i--]);
}
#include<stdio.h>  //palindrome string using pointer
void main(){
char str[30];
char *a,*b;
printf("Enter a string :");
gets(str);
for(a=str ; *a!=NULL ;a++); //here a stores address for 'str' //
  for(b=str, a-- ; a>=b; ) // b stores address for str//
  {
    if(*a==*b) //if we put * sign over a and b then it means the value of
str string variable//
   {
        a--;
        b++;
    }
    else
        break;
  }
  if(b>a)
       printf("\nString is palindrome");
  else
       printf("\nString is Not palindrome");
```

```
}
#include <stdio.h>   // sorting strings without strcmp
#include <string.h>
void main() {
      char *a[] = {"NUTS","apple","nuts","APPLE","OOPS","oops"};
    const char *s1,  *s2;
    int compRes,i,j;
    int length = 9;
    for (i = 0; i < length; i++) {
        for (j = i+1; j < length; j++){
            s1 = a[i];
            s2 = a[j];
            compRes = 0;
            while(*s1 && *s2){
                if(*s1!=*s2){
                    compRes = *s1 - *s2;
                    break;
                }
                ++s1;
                ++s2;
            }
            if (compRes > 0 || (compRes == 0 && *s1)) {
                char* temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
        printf("%s ", a[i]);

    }
    printf("\n");
}
#include <stdio.h>
void main (){
   FILE *fp;
   fp = fopen("file.txt", "w+");
   fputs("This is c programming.", fp);
   fputs("This is a system programming language.", fp);
   fclose(fp);
}
#include <stdio.h>
void main(){
   FILE *fp;
   char str[60];
   fp = fopen("file.txt" , "r");
   if(fp == NULL)
   {
      perror("Error opening file");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) puts(str);
   fclose(fp);
}
#include <stdio.h>
void main (){
   FILE *fp;
   int c;
   fp = fopen("file.txt","r");
   while(!feof(fp))
   {
      c = fgetc(fp);
      printf("%c", c);
   }
   fclose(fp);
}
#include <stdio.h>
void main(){
struct Date{
    int d, m, y;
}f1,f2;
int t, td=0, y1,y2, m1,m2, d1,d2,i;
int md[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
scanf("%d%d%d",&f1.d,&f1.m,&f1.y);
scanf("%d%d%d",&f2.d,&f2.m,&f2.y);
if (f1.y < f2.y || f1.m < f2.m || f1.d < f2.d)
{y1=f2.y; y2=f1.y;m1=f2.m;m2=f1.m;d1=f1.d;d2=f2.d;}
else {y1=f1.y;y2=f2.y;m1=f1.m;m2=f2.m;d1=f1.d;d2=f2.d;}
printf("%d-%d-%d\n",y1,m1,d1);
printf("%d-%d-%d\n",y2,m2,d2);
while(y1>y2){
      if (y2%4==0){
            printf("\n leap year");
            td=td+366;
      }
      else td=td+365;
      y2++;
}
for(i=m2;i<m1;i++){
      if (y2%4==0 && i==1){
            printf("\nlast year is a leap year");
            td=td+1;
      }
      td=td+md[i];
}
td=td+d2-d1+1;
printf("\n%d",td);
}

#include <stdio.h>          //Pascal triangle
long fact (int n)
{
   int c;
   long result = 1;
   for (c = 1; c <= n; c++)
         result = result*c;
   return result;
}
void main()
{
   int i, n, c;
   printf("Enter the number of rows \n");
   scanf("%d",&n);
   for (i = 0; i < n; i++){
      for (c = 0; c <= (n - i - 2); c++) printf(" ");
      for (c = 0 ; c <= i; c++)
         printf("%ld ", fact (i)/( fact (c)* fact (i-c)));
      printf("\n");
   }
}
#include <stdio.h>    // pyramid  1
```

```
#include <conio.h>       //       2 3 2
int main()               //     3 4 5 4 3
{
      int i,j,r,k=1,l=2,m=1;
      scanf ("%d",&r);
      for (i=1;i<=r;i++){
            for(j=1;j<r+1-i;j++) printf(" ");
            for(j=1;j<=l/2;j++) printf("%d ",k++);
            getch();
            m=k-2;
            for(;j<l;j++){printf("%d ",m--);}
            printf("\n");
            l=l+2;
            k=k-1;
         }
}
// Conversion of a decimal number to binary, octal
#include <stdio.h>
void main(void){
    int n, a[20],i=0;
    scanf("%d",&n);
    while (n > 0) {
      a[i++]=n%2;
        n = n / 2;
    }
    i--;
    for(;i>=0;i--)
      printf("%d",a[i]);
}
// decimal number to hexa decimal
#include<stdio.h>
void main(){
    long int d,r,q;    int i=1,j,t;     char h[100];
    scanf("%ld",&d);
    q = d;
    while(q>0){
         t = q % 16;
      if( t < 10)   t = t + 48;
      else          t = t + 55;
      h[i++]= t;
      q = q / 16;
  }
    for(j = i -1 ;j> 0;j--) printf("%c",h[j]);
}
// binary number to decimal
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
void main()
{
    int b, d = 0, c;
    scanf("%d", &b);
    for(c = 0; b > 0; c++){
        d = d + pow(2, c) * (b % 10);
        b = b / 10;
    }
    printf("\nDecimal Equivalent of Binary Number: \t %d\n", d);
}
#include <stdio.h>   // binary number to octal
void main(void){
      long int b, o=0;
      int j=1, r;
      scanf("%ld",&b);
      printf("Binary = %ld\n", b);
      while(b>0){
        r = b % 10;
        o = o + r * j;
        j = j * 2;
        b = b / 10;
      }
      printf("Octal = %lo\n", o);
}
#include<stdio.h> // Octal to Binary Conversion
void main()
{
      long int i=0;
      char octnum[1000];
      gets(octnum);
      while(octnum[i])  {
            switch(octnum[i])
            {
                  case '0' : printf("000");              break;
                  case '1' : printf("001");              break;
                  case '2' : printf("010");              break;
                  case '3' : printf("011");              break;
                  case '4' : printf("100");              break;
                  case '5' : printf("101");              break;
                  case '6' : printf("110");              break;
                  case '7' : printf("111");              break;
                  default : printf("Invalid Octal Digit");  break;
            }
            i++;
      }
}
#include<stdio.h>     // convert binary to octal
void main(){
    char b[100],o[100];
    long int i=0,j=0;
    scanf("%s",b);
    while(b[i]){
      b[i] = b[i] -48;
      ++i;
    }
    --i;
    while(i-2>=0){
    o[j++] = b[i-2] *4 + b[i-1] *2 + b[i];
    i=i-3;
    }
    if(i==1)             o[j] = b[i-1] *2 + b[i];
    else if(i==0)        o[j] =  b[i];
    else                 --j;
    while(j>=0){
      printf("%d",o[j--]);
    }}
#include<stdio.h>   // convert binary to hexadecimal
void main(){
    char b[100],h[100];
    int t;
    long int i=0,j=0;
    scanf("%s",b);
    while(b[i]){
      b[i] = b[i]-48;
      ++i;
    }
    --i;
    while(i-2>=0){
       t =  b[i-3] *8 + b[i-2] *4 +  b[i-1] *2 + b[i];
       if(t > 9) h[j++] = t + 55;
       else      h[j++] = t + 48;
       i=i-4;
    }
    if(i==1)      h[j] = b[i-1] *2 + b[i] + 48;
    else if(i==0) h[j] =  b[i] + 48;
    else --j;
    while(j>=0) printf("%c",h[j--]);
}
#include <stdio.h> // array insertion
void main()
{
   int array[100], position, c, n, value;
   printf("Enter number of elements in array\n");
   scanf("%d", &n);
   printf("Enter %d elements\n", n);
   for (c = 0; c < n; c++) scanf("%d", &array[c]);
   printf("Enter the location where you wish to insert an element\n");
   scanf("%d", &position);
   printf("Enter the value to insert\n");
   scanf("%d", &value);
   for (c = n - 1; c >= position - 1; c--) array[c+1] = array[c];
   array[position-1] = value;
   printf("Resultant array is\n");
   for (c = 0; c <= n; c++) printf("%d\n", array[c]);
}
#include <stdio.h>  // array deletion
void main()
{
   int array[100], position, c, n;
   printf("Enter number of elements in array\n");
   scanf("%d", &n);
   printf("Enter %d elements\n", n);
   for (c = 0; c < n; c++) scanf("%d", &array[c]);
   printf("Enter the location where you wish to delete element\n");
   scanf("%d", &position);
   if (position >= n+1) printf("Deletion not possible.\n");
   else {
      for (c = position - 1; c < n - 1; c++) array[c] = array[c+1];
      printf("Resultant array:\n");
      for (c = 0; c < n - 1; c++) printf("%d\n", array[c]);
   }
}
#include <stdio.h> //bubble sort
void main()
{
      int i, j,n,a[5]={10,15,30,45,70};
    for (i = 0; i < 5; i++)
       for (j = 0; j < 5-i-1; j++)
           if (a[j] < a[j+1]){
                  n=a[j+1];
                  a[j+1]=a[j];
                  a[j]=n;
               }
    for (i = 0; i < 5; i++)
      printf("%d\t",a[i]);
```

```
}
#include <stdio.h> //selection sort
void main()
{
    int i, j,n,t,a[5]={10,15,30,45,70};
    for (i = 0; i < 5-1; i++)    {
        n = i;
        for (j = i+1; j < 5; j++)
          if (a[j] > a[n]) n = j;
          t=a[n];
            a[n]=a[i];
            a[i]=t;
    }
    for (i = 0; i < 5; i++)printf("%d\t",a[i]);
}
#include <stdio.h> //insertion sort
void main()
{
    int i, j,n,t,a[5]={10,15,30,45,70};
    for (i = 0; i < 5; i++)    {
        t = a[i];
        j=i-1;
        while(j>=0 && a[j] > t){
            a[j+1]=a[j];
            j=j-1;
            }
            a[j+1]=t;
    }
    for (i = 0; i < 5; i++)printf("%d\t",a[i]);
}
#include <stdio.h>    //print all unique/frequency elements in array
void main()
{
    int a[10], f[10];
    int s, i, j, c;
    printf("Enter size of array: ");
    scanf("%d", &s);
    printf("Enter elements in array: ");
    for(i=0; i<s; i++)
    {
        scanf("%d", &a[i]);
        f[i] = -1;
    }
    for(i=0; i<s; i++)
    {
        c = 1;
        for(j=i+1; j<s; j++)
        {
            if(a[i] == a[j])
            {
                c++;
                f[j] = 0;
            }
        }
        if(f[i] != 0) f[i] = c;
    }
    printf("\nUnique/frequency/duplicate elements in the array are: ");
    for(i=0; i<s; i++)
        if(f[i] == 1)// if(f[i]!=0) printf("%d ",f[i]); frequency
        printf("%d ", a[i]);  //if(f[i]==0) printf("%d ",a[i]); duplicate
}
```

```
#include <stdio.h> // string length – strlen()
void main()
{
    char a[20];
    int i=0;
    gets(a);
    while(a[i++]!=NULL);
    printf("%d ",i-1);
}
#include <stdio.h> // string copy – strcpy()
void main()
{
    char a[20],b[20];
    int i=0;
    gets(a);
    while(a[i]!=NULL){b[i]=a[i];i++;}
    puts(b);
}
#include <stdio.h>  // string concatenation - strcat()
void main()
{
    char a[20],b[20],c[20];
    int len,i=0,j=0;
    gets(a);
    gets(b);
    while(a[i]!=NULL){c[i]=a[i];i++;}
    while(b[j]!=NULL) c[i++]=b[j++];
    puts(c);
}
#include <stdio.h>  // counting alphabets, numbers, special characters
void main()
{
    char a[20];
    int i=0,alpha=0,digits=0,special=0;
    gets(a);
    while(a[i]!='\0'){
            if((a[i]>='a' && a[i]<='z') || (a[i]>='A' && a[i]<='Z'))
                  alpha++;
            else if (a[i]>='0' && a[i]<='9')
                  digits++;
            else special++;
            i++;
      }
    printf("%d\t%d\t%d",alpha,digits,special);
}
#include <stdio.h> // string comparison - strcmp()
void main()
{
    char a[20],b[20];    int i=0;          // Hello good - 1
    gets(a);    gets(b);                   // Good Hello – (-1)
    while(a[i]!='\0' || b[i]!='\0'){       // hello hello - 0
          if(a[i]!=b[i]) break;
          i++;
    }
    i=a[i]-b[i];
    printf("%d",i);
}
#include<stdio.h> // Delete all occurrences of a character from the string
void main() {
         char a[10],b[10], ch;
         int i, j = 0, len;
```

```
          gets(a);
          ch=getchar();
          while(a[i++]!=NULL);
          len = i;
          for (i = 0; i < len; i++)
                    if (a[i] != ch)
                              b[j++] = a[i];
          b[j] = '\0';
          puts(b);
}
```

Search occurrence of a character in the string.
Replace a character in the string.

Check whether a character is uppercase.	// A=65, Z=90
Check whether a character is lowercase.	// a=97, z=122
Count number of uppercase and lowercase letters.	
Replace lowercase character with uppercase letter.	// difference is 32
Replace uppercase character with lowercase letter.	
Encode a string and display encoded string.	// char-10
Decode a string and display decoded string.	// char+10

```
#include <stdio.h>//Replace a word in string. Good morning-> Good afternoon
#include <string.h>
void main(){
  char s[80];
  char w[10],rw[10],t[10][10];
  int i=0,j=0,k=0,w,p;
  gets(s);
  printf("%s\n",s);
  printf("\nENTER WORD IS TO BE REPLACED\n");
  scanf("%s",w);
  printf("\nENTER BY WHICH WORD THE %s IS TO BE REPLACED\n",word);
  scanf("%s",rw);
  p=strlen(s);
  for (k=0; k<p; k++){
      if (s[k]!=' '){
          t[i][j] = s[k];
          j++;
      }
      else{
          t[i][j]='\0';
          j=0; i++;
      }
  }
  t[i][j]='\0';
  w=i;
  for (i=0; i<=w; i++){
      if(strcmp(s[i],w)==0)
        strcpy(s[i],rw);
      printf("%s ",s[i]);
    }
}
```

Reverse letter in each word of the entered string. // Hello students -> olleH stneduts

```
#include <stdio.h>    // transpose matrix
void main(){
   int m,n,c,d,matrix[10][10],transpose[10][10];
   scanf("%d%d",&m,&n);
   for (c=0;c<m;c++)
      for(d=0;d<n;d++) scanf("%d",&matrix[c][d]);
   for (c=0;c<m;c++)
      for(d=0;d<n;d++) transpose[d][c]=matrix[c][d];
   for (c=0;c<m;c++){
      for (d=0;d<n;d++) printf("%d\t",matrix[c][d]);
      printf("\n");
   }
   for (c=0;c<n;c++){
      for (d=0;d<m;d++) printf("%d\t",transpose[c][d]);
      printf("\n");
   }
}
#include<stdio.h> // print Identity Matrix
void main(){
    int size,row,col;
    scanf("%d",&size);
    for (row = 0; row < size; row++){
        for (col = 0; col < size; col++)
            if (row == col) printf("%d ", 1);
            else printf("%d ", 0);
        printf("\n");
    }
}
#include <stdio.h> // checking whether matrix is an Identity or not
void main(){
    int a[10][10], s, r, c, i;
    scanf("%d",&s);
    for(r=0; r<s; r++)
        for(c=0; c<s; c++)scanf("%d", &a[r][c]);
    i = 1;
    for(r=0; r<s; r++)
        for(c=0; c<s; c++)
            if(r==c && a[r][c]!=1) i = 0;
            else if(r!=c && a[r][c]!=0) i = 0;
    for(r=0; r<s; r++){
        for(c=0; c<s; c++)printf("%d ", a[r][c]);
        printf("\n");
    }
    if(i==1)printf("\nThe given matrix is an Identity Matrix.\n");
    else printf("The given matrix is not Identity Matrix");
}
#include <stdio.h>    //determinant of 2x2 matrix
#define s 2 // Matrix size
void main(){
    int a[s][s], r, c, d;
    for(r=0; r<s; r++)
        for(c=0; c<s; c++)
            scanf("%d", &a[r][c]);
    det = (a[0][0] * a[1][1]) - (a[0][1] * a[1][0]);
    printf("Determinant of matrix A = %ld", det);
}
#include <stdio.h>  //determinant of 3x3 matrix
#define s 3 // Matrix size
void main(){
    int A[s][s], m, n;
    int a, b, c, d, e, f, g, h, i;
    long det;
    for(m=0; m<s; m++)
        for(n=0; n<s; n++) scanf("%d", &A[m][n]);
    a = A[0][0];b = A[0][1];c = A[0][2];
    d = A[1][0];e = A[1][1];f = A[1][2];
    g = A[2][0];h = A[2][1];i = A[2][2];
    det = (a*(e*i-f*h))-(b*(d*i-f*g))+(c*(d*h-e*g));
    printf("Determinant of matrix A = %ld", det); }
```

```
#include<stdio.h>  //puzzle
void main(){
  int i = 10;
  int c = 10;
  switch(c)   {
    case i: // not a "const int" expression
         printf("Value of c = %d", c);
         break;
  } } // [Error] case label does not reduce to an integer constant
#include<stdio.h>  //puzzle
void main(void){
  int i = 0;
  for(i = 0; i < 3; i++)  {
    printf("loop ");
    continue;
  } } // Output: loop loop loop
#include<stdio.h>  //puzzle
void main(void){
  int i = 0;
  while(i < 3){
    printf("loop"); /* printed infinite times */
    continue;
    i++; /*This statement is never executed*/
} } // infinite loop
#include <stdio.h>  //puzzle
void main(){
   float x = 1.1;
   switch (x)   {
       case 1.1: printf("Choice is 1"); break;
       default: printf("Choice other than 1, 2 and 3"); break;
   }
} // [Error] switch quantity not an integer
[Error] case label does not reduce to an integer constant
#include <stdio.h>  //puzzle
void main(){
   int i = 0;
   while ( 1 )    {
      printf( "%d\n", ++i );
      if (i == 5)     break; // Used to come out of loop
}
} // Output: 1 \n 2 \n 3 \n 4 \n 5
# include <stdio.h>  //puzzle
void main() {
  int i = 0;
  while ( 0 )   {     // This line will never get executed
    printf( "%d\n", ++i);
    if (i == 5)       break;
  }
} // Output: Nothing printed
#include <stdio.h>  // There is no break in all cases
void main(){
   int x = 2;
   switch (x)   {
       case 1: printf("Choice is 1\n");
       case 2: printf("Choice is 2\n");
       case 3: printf("Choice is 3\n");
       default: printf("Choice other than 1, 2 and 3\n");
   }
} // Output: case 2, 3, and default are executed
```

```
#include <stdio.h> // A program with variable expressions in labels
void main(){
    int x = 2;
    int arr[] = {1, 2, 3};
    switch (x)    {
        case arr[0]: printf("Choice 1\n");
        case arr[1]: printf("Choice 2\n");
        case arr[2]: printf("Choice 3\n");
    }
} // [Error] case label does not reduce to an integer constant
#include <stdio.h> //puzzle
void main(){
   int x = 1;
   switch (x)   {
       x = x + 1; // Statements before all cases are never executed
       case 1: printf("Choice is 1");               break;
       case 2: printf("Choice is 2");                break;
       default: printf("Choice other than 1 and 2"); break;
   }
} // Output: Choice is 1
#include <stdio.h> //puzzle
void main(){
   int x = 1;
   switch (x)   {
       case 2: printf("Choice is 1"); break;
       case 1+1: printf("Choice is 2"); break;
   }} // [Error] duplicate case value
#include <stdio.h> // C program to illustrate using range in switch void
void main(){
    int i, arr[] = { 1, 5, 15, 20 };
    for ( i = 0; i < 4; i++) {
        switch (arr[i]) {
        case 1 ... 6: printf("%d in range 1 to 6\n", arr[i]); break;
        case 19 ... 20: printf("%d in range 19 to 20\n", arr[i]);  break;
        default: printf("%d not in range\n", arr[i]);  break;
        }
    }
} // Output: Prints all 3 case statements
#include<stdio.h> //puzzle
void main() {
    if(!printf("Hello"))
        printf("Hello");
    else
        printf("World");
} // Output: HelloWorld
#include <stdio.h> //puzzle
void main(){
    int i, n = 20;
    for (i = 0; i < n; i-- or n-- or n++ or i++) printf("*");
} // Output: i--, n++ infinite loop, i++, n-- prints 20 times *
#include<stdio.h>
void main()  {
          int a=1;
          switch(a)          {
                  case '1':   printf("ONE\n");  break;
                  case '2':  printf("TWO\n");  break;
                  default:   printf("NONE\n");
          }
} // No Output - case values are in char, but a is int
```

```
#include<stdio.h> //puzzle
void main(){
      int b=1;
      switch(b) {
           int b=20; // not executed
           case 1: printf("b is %d\n",b);       break;
           default: printf("b is %d\n",b);       break;
      }
}  // Output: b is 1
void main(){                                        //puzzle
   int x = 2, y = 5;
   (x & y)? printf("True ") : printf("False ");
   (x && y)? printf("True ") : printf("False ");
} // Output: False True
void main(){                                        //puzzle
   int x = 19;
   printf ("x << 1 = %d\n", x << 1);
   printf ("x >> 1 = %d\n", x >> 1); } // Output: 38 9
void main(){                                        //puzzle
   int x = 19;
   (x & 1)? printf("Odd"): printf("Even"); // Output: Odd
   (x & 0)? printf("Odd"): printf("Even"); // Output: Even
} // Output: EvenOdd
void main(){                                        //puzzle
   unsigned int x = 1;
   printf("Signed Result %d \n", ~x);
   printf("Unsigned Result %ud \n", ~x);
} // Output:  Signed Result -2
// Unsigned Result 4294967294d

printf("%%");Output -> -> %

printf("%d",printf("%s","hello")); Output -> hello5

char a[50]; printf("%d", scanf("%s",a)); Output -> 1

printf(5+"hello world"); Output -> world

printf("%c",5["hello world"]); Output -> No output

printf("%c",["hello world"]5); Output -> Error

int a; printf("%d",1 + ++a); Output -> 2

int a=1,2,3; Output -> Error

int a; a=1,2,3; Output -> No output

int a; a=1,2,3; printf("%d",a); Output -> 1

int a; a=(1,2,3); printf("%d",a); Output -> 3

int x=10, y=10; printf("%d", sizeof(x==y)); Output -> 4
```

Happy / Sad Number Calculation
In recreational mathematics, the "happy numbers" and "sad numbers" are the integers that produce particular sequences when you repeatedly square the number's digits and compute the sum of those squares. Integers that stabilize at 1 are called "happy numbers" and integers that fall into the loop are called "sad numbers". For the example, Positive Integer = 103
Sum of Digits of 103 = $1^2 + 0^2 + 3^2 = 1 + 0 + 9 = 10$, $10 = 1^2 + 0^2 = 1 + 0 = 1$, so 103 is a happy number

```
#include <stdio.h>        //GCD or HCF using recursion
int hcf(int n1, int n2);
void main(){
   int n1, n2;
   scanf("%d %d", &n1, &n2);
   printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
}
int hcf(int n1, int n2){
    if (n2 != 0) return hcf(n2, n1%n2);
    else return n1;
}
```

```
#include <stdio.h>  // find Sum of Digits of a Number using Recursion
int sum (int a);
void main(){
    int num, result;
    scanf("%d", &num);
    result = sum(num);
    printf("Sum of digits in %d is %d\n", num, result);
}
int sum (int num){
    if (num != 0) return (num % 10 + sum (num / 10));
    else return 0;
}
```

```
#include <stdio.h> //  power of a number using recursion
int power(int n1, int n2);
void main(){
    int base, powerRaised, result;
    scanf("%d",&base);
    scanf("%d",&powerRaised);
    result = power(base, powerRaised);
    printf("%d^%d = %d", base, powerRaised, result);
}
int power(int base, int powerRaised){
    if (powerRaised != 0)
        return (base*power(base, powerRaised-1));
    else
        return 1;
}
```

```
#include<stdio.h>  //Reverse of given number using recursion
 void main(){
   int num,x;
   scanf("%d",&num);
   x=rev(num);
   printf("Reverse of given number is: %d",x);
 }
 int rev(int num){
   static sum,r;
   if(num){
     r=num%10; sum=sum*10+r; rev(num/10);
   }
   else return 0;
   return sum;
 }
```

```
#include<stdio.h> // number writing and reading from file
 void main(){
   FILE *fp;
   int i=10, j=20, k=3, num;
   fp = fopen ("test.dat","w");
   putw(i,fp);
   putw(j,fp);
   fclose(fp);
   fp = fopen ("test.dat","r");
   do {
      num= getw(fp);
      if(num!=-1) printf("Data in test.dat file is %d \n", num);
   }while(num!=-1);
   fclose(fp);
}

#include <stdio.h> // character writing and reading from file
int main()
{
   char ch;
   FILE *fp;
   if (fp = fopen("test.dat", "r")){
     ch = getc(fp);
     while (ch != EOF){
        putc(ch, stdout);
        ch = getc(fp);
     }
     fclose(fp);
     return 0;
   }
   return 1;
}

#include<stdio.h> // string writing and reading from file
#include<string.h>
#include<stdlib.h>
 void main(){
    FILE *fp ;
    char data[50];
    fp = fopen("test.dat", "w") ;
    if ( fp == NULL )
    {
        printf( "Could not open file test.dat" ) ;
        exit(1);
    }
    printf( "\n Enter some text from keyboard");
    while (strlen(gets(data))>0){
        fputs(data, fp) ;
        fputs("\n", fp) ;
    }
    fclose(fp);
    fp = fopen("test.dat", "r");
      while(fgets(data,50,fp) != NULL)printf( "%s" , data ) ;
    fclose(fp) ;
}
```

```
# include <stdio.h> // character reading from file
int main(){
   FILE *fp ;
   char c ;
   fp = fopen ( "test.dat", "r" ) ;
   if ( fp == NULL )
   {
     printf ( "Could not open file test.dat" ) ;
     return 1;
   }
   while ( 1 )
   {
     c = fgetc ( fp ) ;
     if( feof(fp) )    break ;
     printf ( "%c", c ) ;
   }
   fclose ( fp ) ; // Closing the file
}

# include <stdio.h>  // binary write and read
struct date{
   int n1, n2, n3;
};
int main(){
   int n;
   struct date num;
   FILE *fptr;
   if ((fptr = fopen("test.dat","wb")) == NULL){
       printf("Error! opening file");
       return 1;
   }
   for(n = 1; n < 5; n++){
      num.n1 = n;   num.n2 = 2;   num.n3 = 2019;
      fwrite(&num, sizeof(struct date), 1, fptr);
   }
   fclose(fptr);
   fptr = fopen("test.dat","rb");
   fseek(fptr, sizeof(struct date), SEEK_SET); // SEEK_END, SEEK_CUR
   for(n = 1; n < 5; ++n)    {
      fread(&num, sizeof(struct date), 1, fptr);
      printf("\n n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);
   return 0;
}

# include <stdio.h>  //removing file
int main(){
      remove("test.dat");
}
```

```c
#include <stdio.h> // example for malloc
#include <stdlib.h>
void main(){
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL){
        printf("Error! memory not allocated."); exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
}
#include <stdio.h> //example for calloc
#include <stdlib.h>
void main(){
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
}
#include <stdio.h> //example for realloc

#include <stdlib.h>
void main(){
    int *ptr, i , n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i) printf("%u\n",ptr + i);
    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2 * sizeof(int));
    printf("Addresses of newly allocated memory: ");
    for(i = 0; i < n2; ++i) printf("%u\n", ptr + i);
free(ptr);}
```