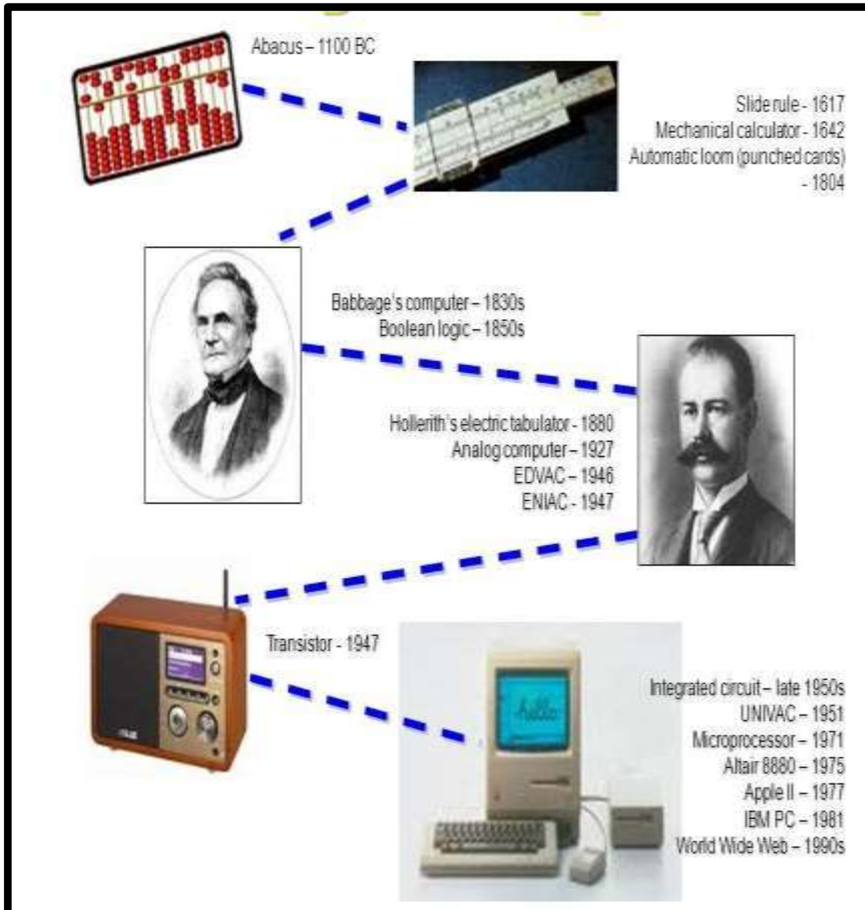
Module 1: INTRODUCTION TO COMPUTER HARDWARE AND SOFTWARE

Swetha M S
Asst. Professor
Dept. of ISE
BMSIT&M

What is computer?

A Computer is device that can automatically performs a set of instructions. The computer takes as input these instructions as a single unit, uses them to manipulate the data, and outputs the results in user-specified ways. The processing is fast, accurate and consistent, and is generally achieved without significant human intervention.

History of Computer



The Mechanical Age	1600-1900	
The Abacus 	Adding Machine 	Analytical Engine 
1900-1945		
	Mark-1 	

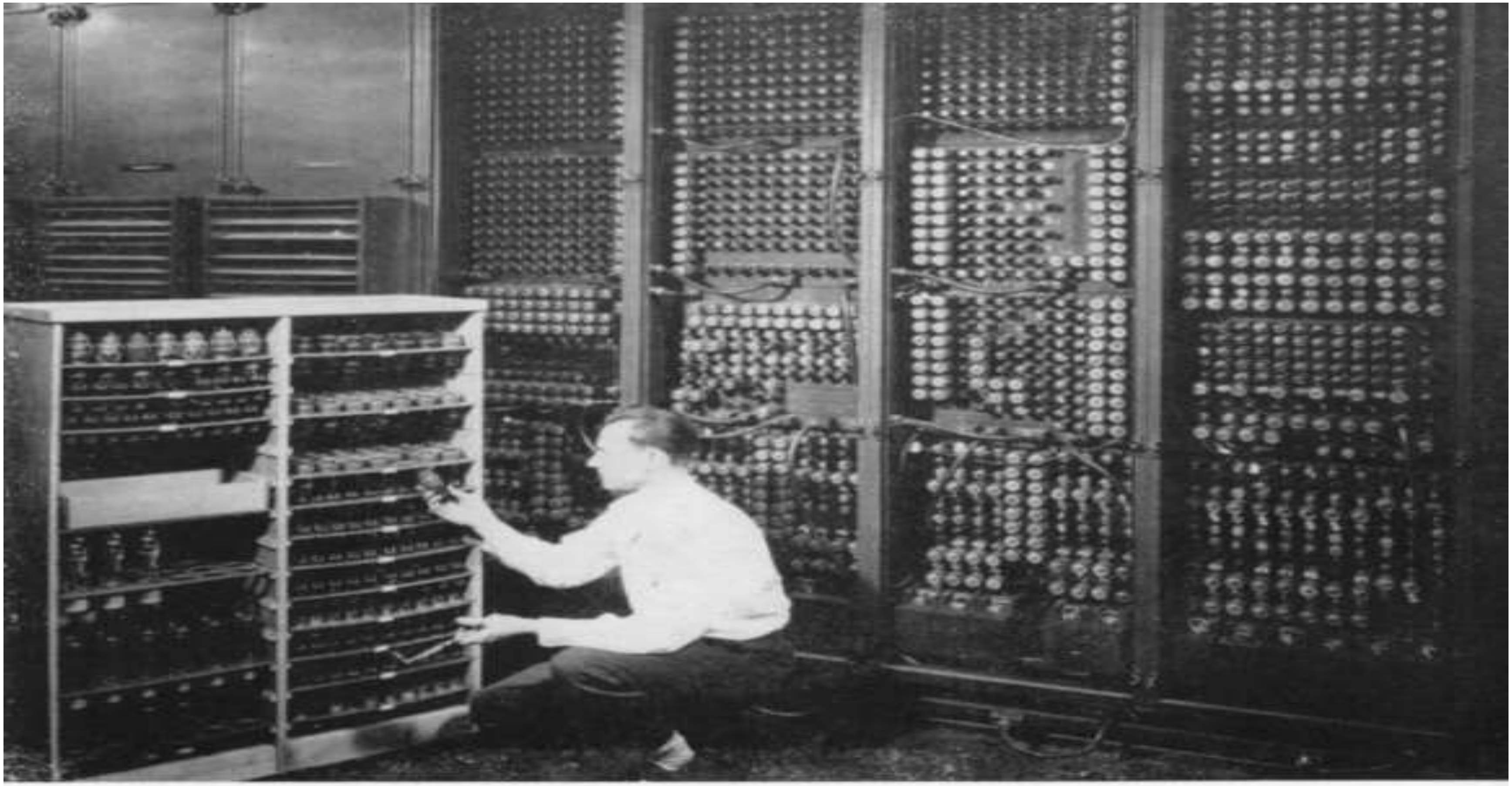
COMPUTER GENERATIONS

Generation	Based on	Other Features
First.	Vacuum tubes	Magnetic drums for memory
Second	Transistors	Magnetic cores, disks, punched cards and printouts
Third.	Integrated circuits (ICs)	Keyboard, monitor and operating system
Fourth	Microprocessors	Networking
Fifth	ULSI Nano technology.	Mainly unclear

Vacuum Tubes: The First Generation

- Memory requirements were met by magnetic drums (forerunner of today's hard disk).
- Because of the size of vacuum tubes, first generation computers took up a lot of space.
- They also consumed enormous amounts of power and generated a lot of heat. In spite of housing these computers in air-conditioned enclosures, frequent breakdowns were common.
- The ENIAC used 18,000 vacuum tubes, occupied 1800 sq. ft. of room space and consumed 180KW of power.
- Machines of this generation were prohibitively expensive to buy and maintain.
- First-generation computers were programmed using a first-generation language-machine language.
- Program input was provided by punched cards and output was obtained on paper.
- First-generation computers were only used for scientific work and were not deployed commercial

Vacuum Tubes



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Transistors: The Second Generation

- Compared to vacuum tubes, transistors were faster, smaller and consumed less power smaller magnetic cores also replaced the first-generation magnetic drums.
- Even though transistor generated less heat, second-generation computers still needed air-conditioning.
- The input-output mechanism however remained largely unchanged.
- Second-generation computers were programmed using a symbolic or assembly language.
- The computers also implemented the stored program concept which allowed both program and data to reside in memory.

Transistors: The Second Generation


1402 Card Read Punch
1407 Console
1401 CPU
729 Tape Drive
1403 Line Printer

Integrated Circuits: The Third Generation

- By virtue of miniaturization, computers consequently got smaller, cheaper and energy efficient. For these reasons, they could be seen in several medium-sized organizations.
- This generation adopted a keyboard and monitor to interact with the user.
- Memory capacity increased substantially and the magnetic hard disk was used for secondary storage.
- Third-generation computers also had an operating system, which is a special program meant to control the resources of the computer.
- By virtue of a feature known as time sharing, the computer could run programs invoked by multiple users.
- The existing programming languages were supplemented by BASIC, C, C++ and Java.

Integrated Circuits: The Third Generation

The Microprocessor: The Fourth Generation

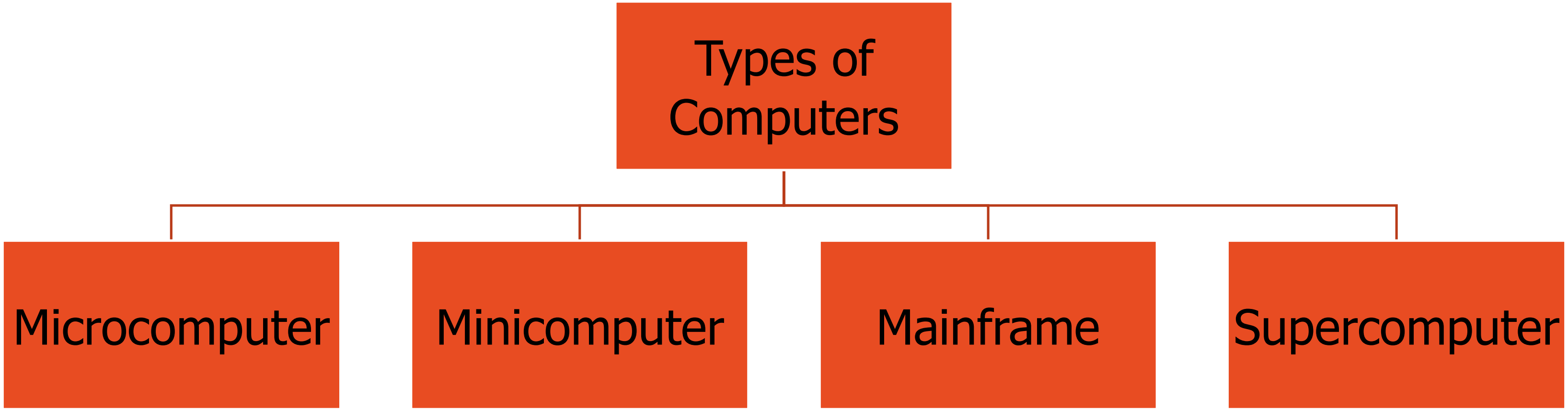
- The integration of components went several steps ahead. Using LSI and VLSI technology, it is now possible to have the entire CPU, its associated memory and input/output control circuitry on a single chip.
- Intel introduced the 4004 microprocessor in 1971 and improvement in the usual parameters (like speed, heat generation, size, etc.) continues at a frenetic pace to this day.
- Microprocessors have invaded our homes to drive desktops, laptops, smartphones,microwave ovens and washing machines.
- Laptops and smartphones offer gigabytes (GB) of memory compared to a few megabytes (MB) that were available in the early days of this generation.
- Operating systems have moved from the rudimentary MSDOS to a mouse based Graphical User Interface (GUI) like Windows. More advanced systems like Linux are now available for desktops and laptops, and a variant of it (Android) powers most of our smartphones.

The Microprocessor: The Fourth Generation

Artificial Intelligence: The Fifth Generation

- The fifth generation represents a vision of the computers of the future. The conventional parameters of computing (speed, size, energy consumption, VLSI to UL.SI, etc.) would continue to improve path-breaking changes in the way we use computers are also expected.
- Fifth-generation systems should be capable of producing human-like behaviour. These systems expected to interact with users in natural language and learn from experience. Speech recognition and speech output should also be possible with these systems.
- Computer speeds need to make an exponential jump, a feat that would be possible using quantum computers.
- Computers must be able to perform parallel processing so that multiple processors concurrently handle different aspects of a problem.
- Neural networks and expert systems have to be developed. These applications would be able to make decisions and advise humans by analysing data using human-like intelligence but without using the services of an expert.

Types of Computers



1. Microcomputer

- Can be classified into:
  - Desktop PCs
    - sits on desks, rarely moved, large and bulky.
    - Memory capacity, graphics capacity and software availability vary from one computer to another Used both for business and home applications

Microcomputer

- Portable PCs
  - Can be moved easily from place to place
  - Weight may varies
  - Small PCs are popular known as laptop
  - Widely used by students, scientist, reporters, etc

Microcomputer Model

Desktop

Laptop

Notebook

Subnotebook

Palmtop

Microcomputer

- **Advantages**
  - Small size
  - Low cost
  - Portability
  - Low Computing Power
  - Commonly used for personal applications

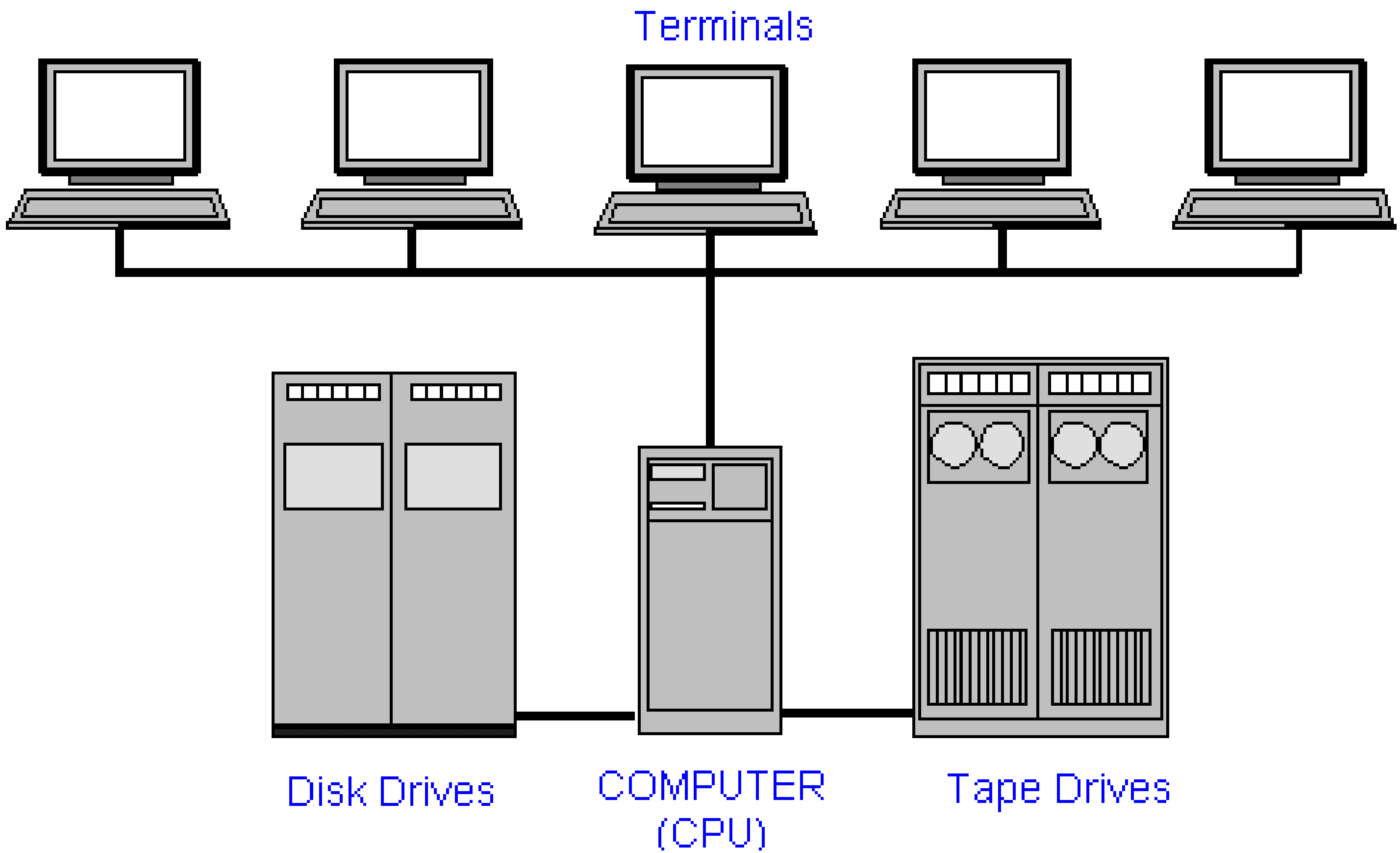
- **Disadvantages**
  - Low processing speed

Uses of Microcomputer

- Word Processing
- Home entertainment
- Home banking
- Printing

2. Minicomputer

- Medium sized computer
- Also called the minis
  - e.g. IBM36, HP9000, etc
- Computing power lies between microcomputer and mainframe computer


Terminals
Disk Drives
COMPUTER
(CPU)
Tape Drives

MiniComputer

- Characteristics
  - Bigger size than PCs
  - Expensive than PCs
  - Multi-User
  - Difficult to use
  - More computing power than PCs
  - Used by medium sized business organizations, colleges, libraries and banks.

Uses of Minicomputer

- Control of Automated Teller Machine (ATMs)
- Hospital patients registration
- Inventory Control for supermarket
- Insurance claims processing
- Small bank accounting and customer details tracking

Minicomputer

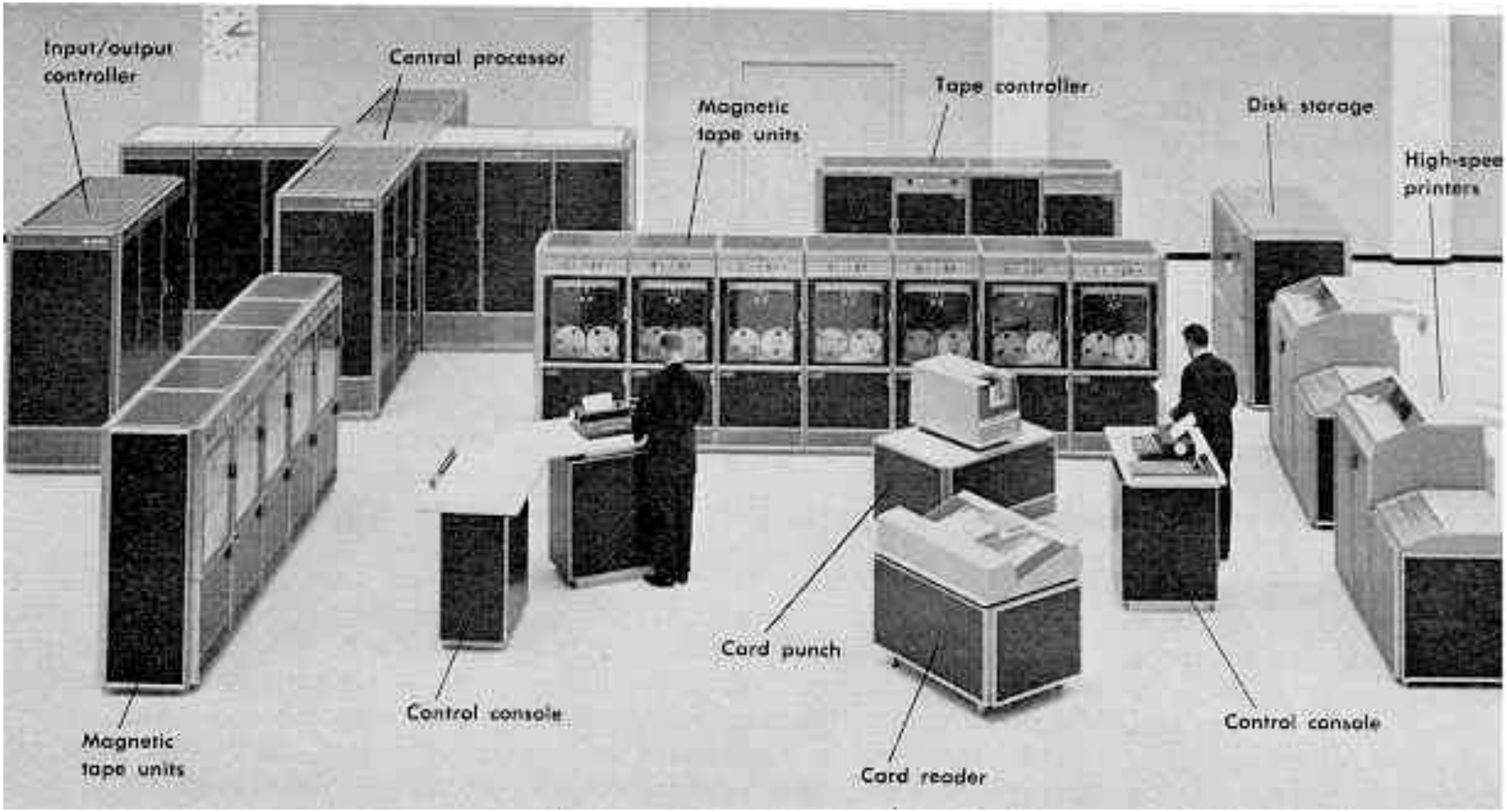
- **Advantage**
  - Cater to multiple users
  - Lower costs than mainframes
- **Disadvantage**
  - Large
  - Bulky

3. Mainframe

- Known as **enterprise servers**
- Occupies entire rooms or floors
- Used for centralized computing
- Serve distributed users and small servers in a computing network

Main Frame

- Large, fast and expensive computer
- Cost millions of dollar
  - e.g. IBM3091, ICL39, etc
- Characteristics:
  - Bigger in size than minicomputers Very expensive
  - Support a few hundred users simultaneously (Multi-Users)
  - Difficult to use
  - More computing power than minicomputers
  - Have to be kept in a special air-conditioned room
  - Used in big business organizations and government departments


Input/output controller
Central processor
Magnetic tape units
Tape controller
Disk storage
High-spee
printers
Card punch
Control console
Magnetic tape units
Card reader
Control console

Mainframe

- **Advantage**
  - Supports many users and instructions
  - Large memory
- **Disadvantage**
  - Huge size
  - Expensive

Supercomputer

- Fastest and expensive
- Used by applications for molecular chemistry, nuclear research, weather reports, and advanced physics
- Consists of several computers that work in parallel as a single system

Super Computer

- **Advantage**
  - Speed
- **Disadvantage**
  - Generate a large amount of heat during operation

Information Processing System

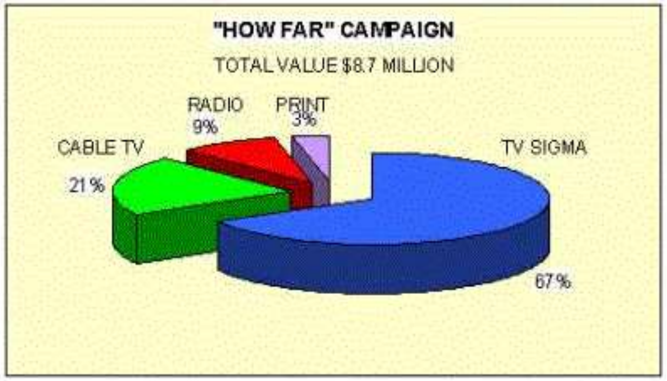
- DATA is a collection of independent and unorganized facts.
- INFORMATION is the processed and organized data presented in a meaningful form.
- DATA PROCESSING is the course of doing things in a sequence of steps.

Information Processing System

- COMPUTER is an electronic machine that follows a set of instructions in order that it may be able to accept and gather data and transform these into information.

PROCESSING SYSTEM

DATA



INFORMATION

Functions of an Information Processing System

1. It accepts and gather data. (INPUT)

1. It processes data to become information. (PROCESSING)

2. It stores data and information. (STORE)

3. It presents information. (OUTPUT)

Three Major Components of an Information Processing System

- HARDWARE is the tangible part of a computer system.
- SOFTWARE is the non-tangible part that tells the computer how to do its job.
- PEOPLEWARE refer to people who use and operate the computer system, write computer programs, and analyze and design the information system.

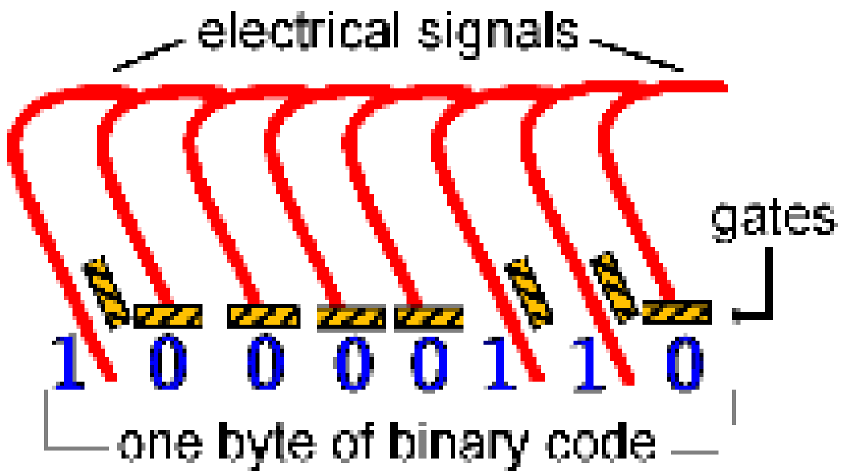
Basic Units of Measurement

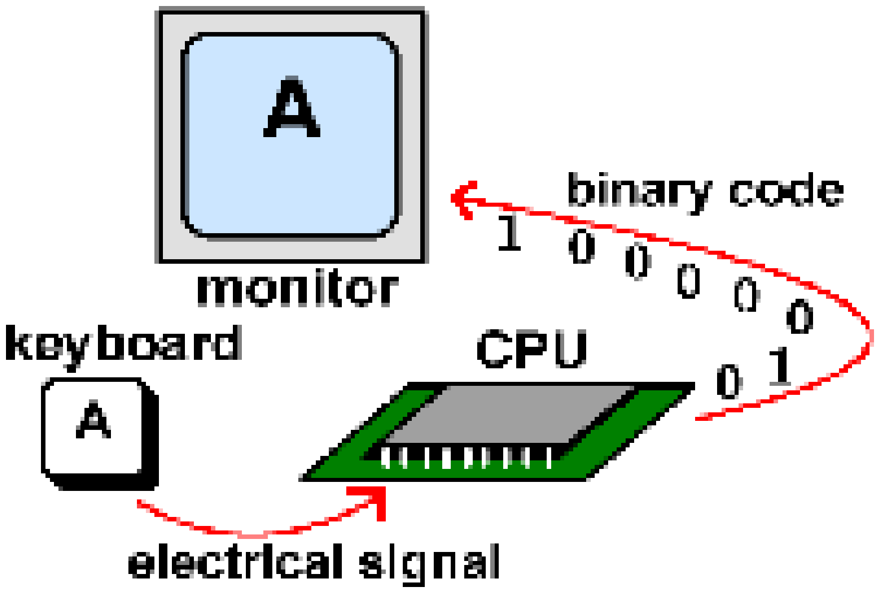
- BIT is a unit of information equivalent to the result of a choice between only 2 possible alternatives in the binary number system.

- BYTE is a sequence of 8 bits (enough to represent one character of alphanumeric data) processed as a single unit for information.

Basic Units of Measurement

- A byte can be used to represent a single character, which can be:
  - A letter
  - A number
  - A special character or symbol, or
  - A space


electrical signals
gates
1 0 0 0 0 1 1 0
one byte of binary code


A
monitor
binary code
1 0 0 0 0 0 0 1
keyboard
A
CPU
electrical signal

BITS, BYTES AND WORDS

Unit	Equivalent to	Remarks
1 kilobyte (KB)	1024 bytes	Space used by 10 lines of text.
I megabyte (MB)	1024 kilobytes	Memory of the earliest PCs
1 gigabyte (GB)	1024 megabytes	Storage capacity of a CD-ROM
I terabyte (TB)	1024 gigabytes	Capacity of today's hard disks.
1 petabyte (PB)	1024 terabytes	Space used for rendering of film Avatar

INSIDE THE COMPUTER

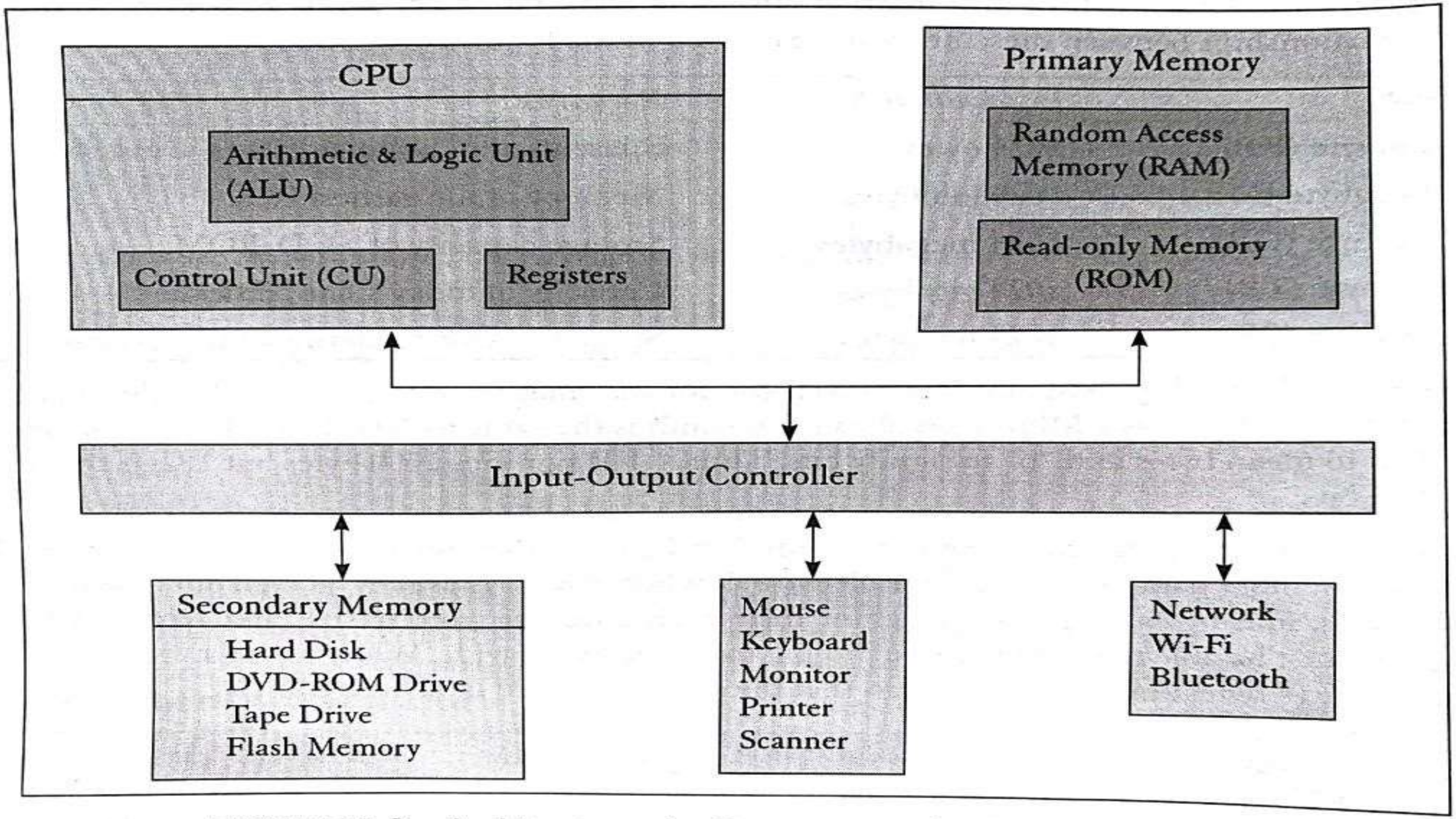


FIGURE 1.3 Architecture of a Computer with a Single Processor

THE CENTRAL PROCESSING UNIT (CPU)

- The CPU has evolved from a bulky vacuum tube based unit of the 1940s to a modern 5cm square chip that is commonly called the microprocessor, or simple processor. It comprises the following components
- Arithmetic and Logic Unit (ALU)
- Control Unit (CU)
- Special purpose registers
- A clock

PRIMARY MEMORY

The primary memory which includes the following types:

- Random Access Memory (RAM-SRAM and DRAM)
- Read Only Memory (ROM, PROM, EPROM, EEIROM)
- Cache Memory (Li, L2 and L3)
- CPU Registers

SECONDARY MEMORY

The last couple of decades have seen the emergence of multiple types of storage devices.

- Hard disk including the portable disk (500 GB to 4 TB).
- Magnetic tape (20 TB).
- CD-ROM (700 MB-less than 1 GB).
- DVD-ROM (4.7 GB and 8.5 GB).
- Blu-ray disk (27 GB and 50 GB).
- Flash memory based on the EEPROM (1 GB to 128 GB).
- The obsoleted floppy disk (1.2 MB and 1.44 MB).

The Hard Disk

- Every disk contains a ***spindle*** that holds one or more ***platters*** made of non-magnetic material like glass or aluminium (Fig. 1.4). Each platter has two surfaces coated with magnetic material.
- Information is encoded onto these platters by changing the direction of magnetization using a pair of read-write heads available for each platter surface.
- Eight surfaces require eight heads; they are mounted on a single arm and cannot be controlled individually.
- Each surface is composed of a number of concentric and serially numbered ***tracks.***
- There are many tracks bearing the same track number as there are surfaces. This can then visualize a ***cylinder*** comprising all tracks bearing the same number on each disk surface.
- Thus, there will be as ***cylinders*** in the disk as there are tracks on each usable surface.

➢ Each track is further broken into ***sectors or blocks***. So, if each track has 32 blocks and a disk has eight surfaces, then Up 256 blocks per cylinder.

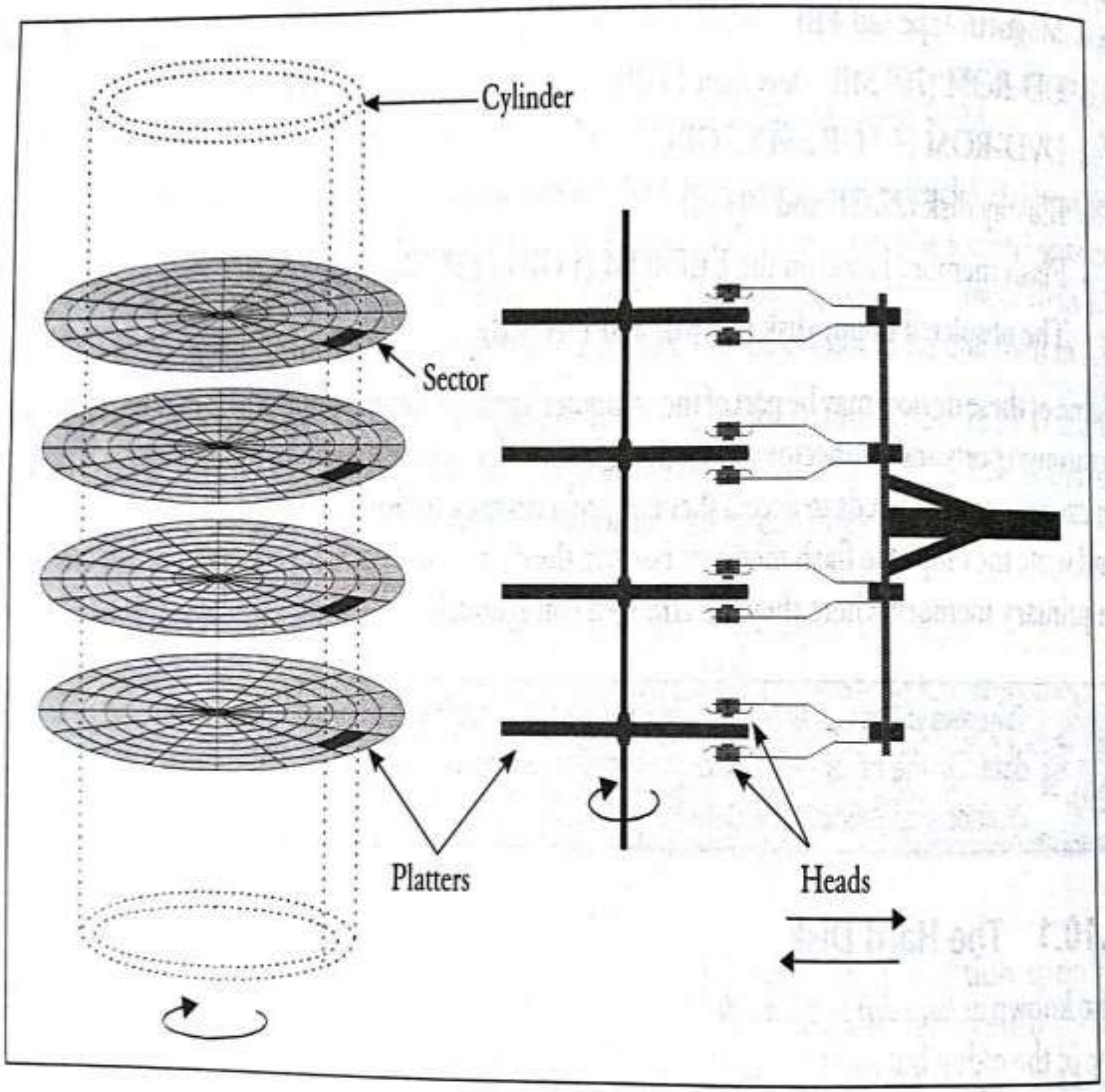
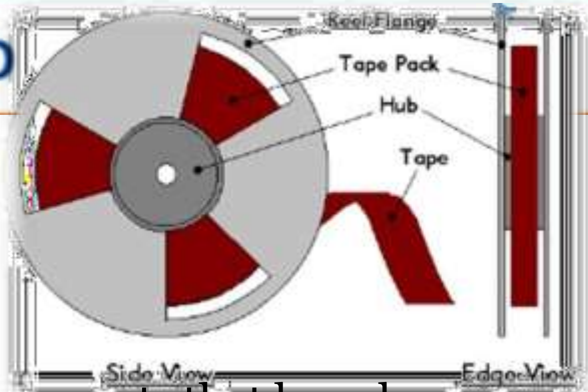


FIGURE 1.4 The Hard Disk



Magnetic Tape

- The age-old magnetic tape is still around thanks to the enhancements that have been made to this device.
- The basic technology has not changed though; the tape is made of a plastic film with one side coated with magnetic material.
- Current technology supports capacities of 1 TB or more, but 200 TB tapes are expected to be launched in the near future.
- The device is portable though because a separate tape drive is required, and most computers don't have one.
- Data are read from and written to the tape using a read-write head and an erasure head.
- The write operation is preceded by the erasing operation. The data access is sequential. To locate a file, the tape has to be rewound before a sequential search can begin.

Optical Disks: The CD-ROM, DVD-ROM

- Non-volatile read-only memory, which we saw in the ROM family (including PROM, EPROM and EEPROM), is also available on optical disks. These disks, comprising mainly the CD-ROM and DVD-ROM, can hold large volumes of data (700 MB to 8.5 GB) on inexpensive media.
- CD-R, DVD-R – Data can be recorded only once, CD-RW, DVD-RW – Data can be recorded multiple times.
- The optical drive uses three motors for the following functions: operating the tray, spinning the disk and guiding the laser beam.

Flash Memory

- They are portable, need little power and are quite reliable.
- The *memory stick* or *pen drive* is the most common type of flash memory used on the computer.
- The ***solid state disk (SSD)*** a bigger device meant to replace the traditional magnetic hard disk. Many small laptops (like Chrome books) have the operating system and a small set of programs stored on this *online* device.
- The third device, the ***magnetic card***, is used mainly in cameras, but using adapters, they can connect to the USB port as well.
- The most popular form of this device is the micro-SD card, which is available in SDHC and SDXC flavours. The SD card offer high capacities that can go up to 128 GB.

Floppy Diskette

- The *floppy diskette* was once the only form of portable storage that could be carried in the pocket.
- A read/write head actually makes contact with this disk while it is rotating.
- The floppy was available in two sizes (5.25" and 3.5"), offering capacities of 1.2 MB and 1.44 MB (yes, MB not GB), respectively.

PORTS AND CONNECTORS

1. *Universal Serial Bits (USB)*
2. *Serial port*
3. *Parallel port*
4. *Video Graphics Array (VGA) port*
5. *digital video interface (DVI)*
6. *PS(Personal System)/2 port*
7. *High Definition Multimedia Interface (HDMI)*

FIGURE 1.6 Common Ports

INPUT DEVICES

1. **The Keyboard**
2. **Pointing Devices**
3. **The Scanner**

OUTPUT DEVICES

1. **The Monitor**
2. **Impact Printers**
   - ***Dot-matrix Printer***
   - ***Daisy-wheel Printer***
   - ***Line Printer***
3. **Non-Impact Printers**
   - ***Laser Printer***
   - ***Ink-jet Printer***
4. **Plotters**

Classification of Printers

Printer

- Impact Printer
  - Daisy Wheel
  - Dot-Matrix
- Non-Impact Printer
  - Ink-jet
  - Thermal
  - Laser





COMPUTERS IN A NETWORK

- Interconnection of computer is called a **computer network**.
- Different ways of connecting computers in network is called as **network topology**.

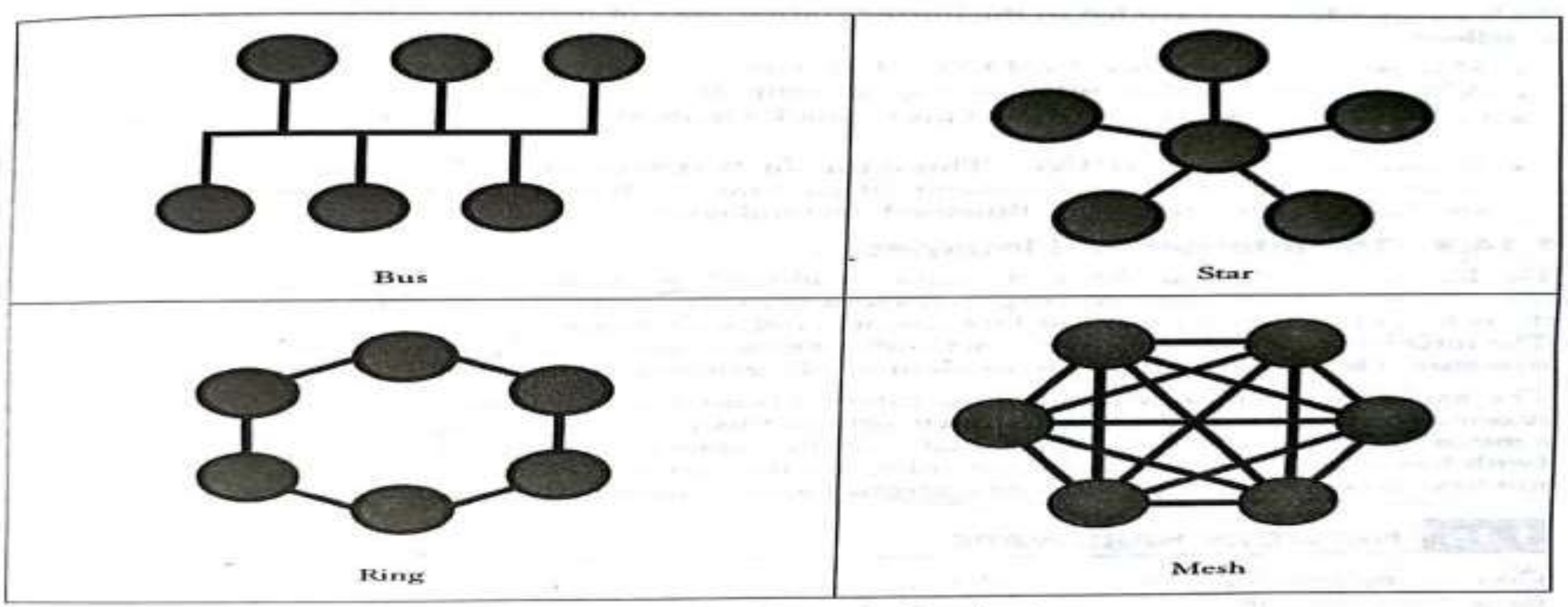


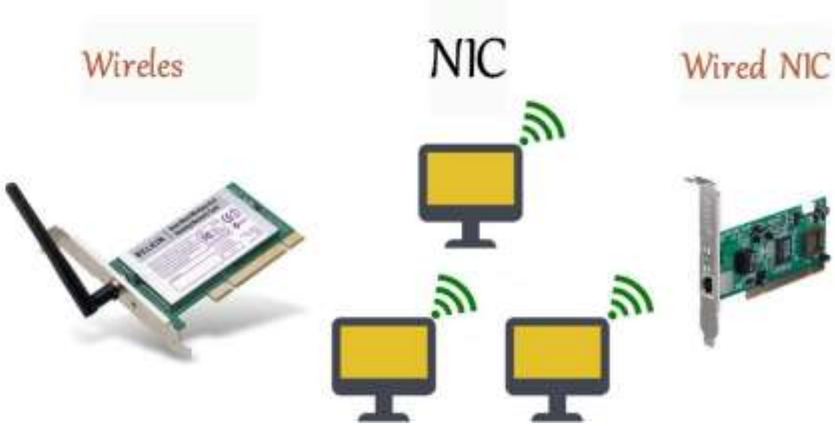
FIGURE 1.7 Network Topologies

Network Types

- Local Area Network (LAN)
- Wide Area Network (WAN)
- Technology advances have led to the birth of other types of networks
- Metropolitan Area Network (MAN)
- Campus Area Network (CAN)
- Personal Area Network (PAN)
- The Internet and internet

NETWORK HARDWARE

- **Network Interface Card**
- **Hub and Switch**
- **Bridge and Router**



WHY COMPUTERS NEED SOFTWARE

Software is a collection of code that drives a computer to perform a related group of tasks.

SOFTWARE TYPES

- **System software**
  - **Basic Input Output System (BIOS)**
  - **Operating system**
  - **Device driver**
  - **Compilers and associated programs**
- **Application software**
  - **Office software**
  - **Database software**
  - **Communications software**
  - **Entertainment software**

HISTORY OF C

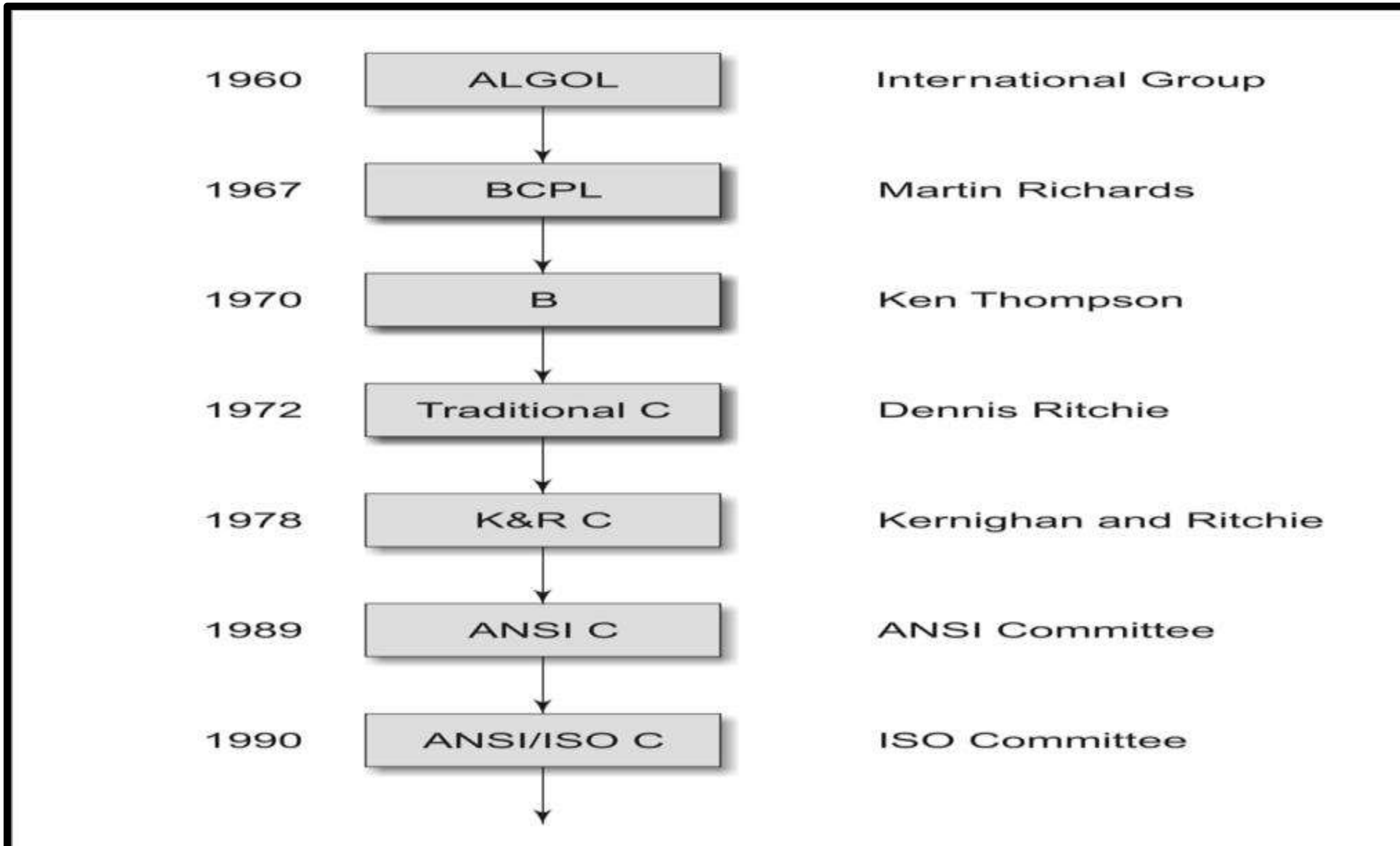


Fig. 1.1 *History of ANSI C*

BASIC STRUCTURE OF C PROGRAMS

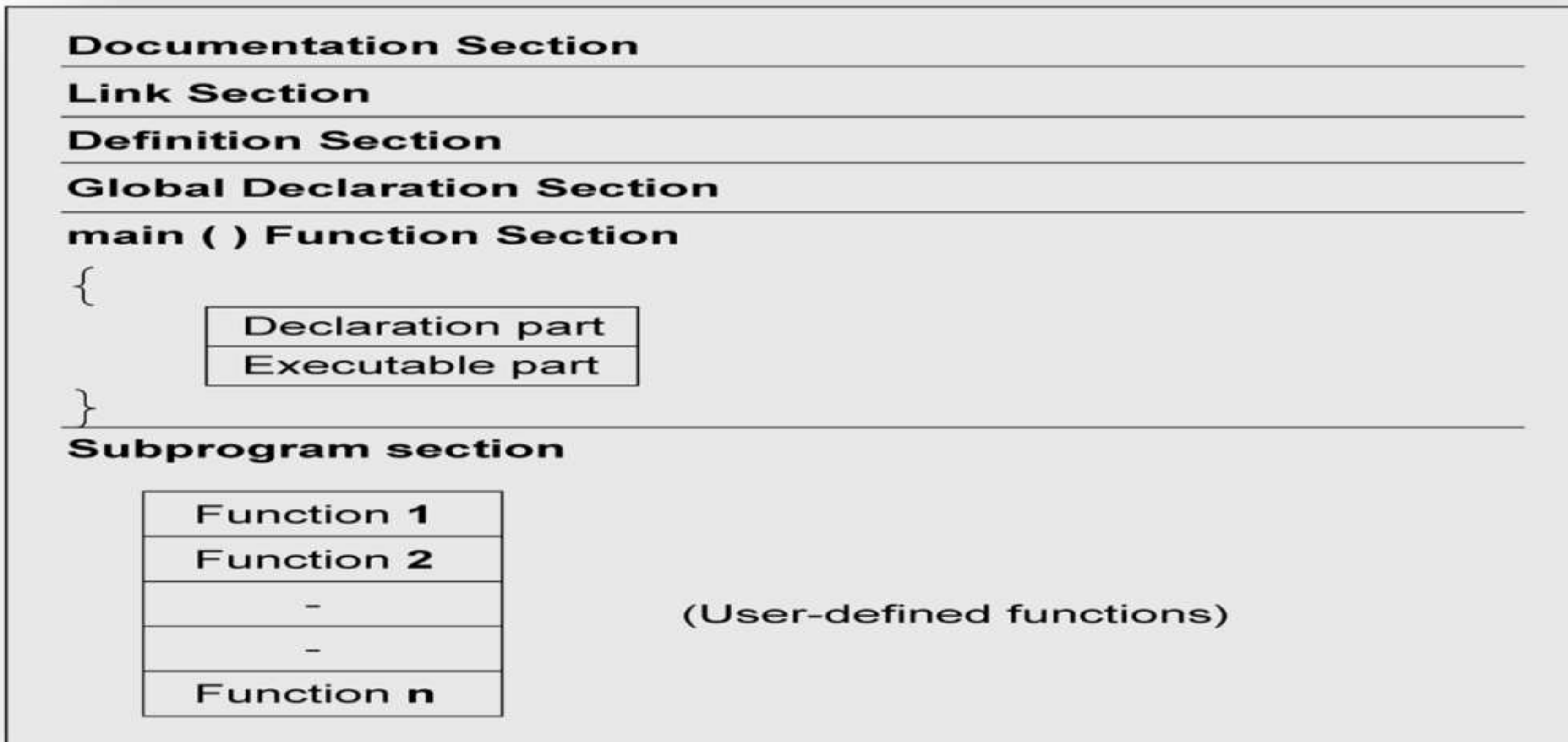


Fig. 1.9 *An overview of a C program*

C Syntax and Hello World

#include inserts another file. ".h" files are called "header" files. They contain stuff needed to interface to libraries and code in other ".c" files.

This is a comment. The compiler ignores this.

```
#include <stdio.h>
/* The simplest C Program */
Int main ( )
{
  printf("Hello World\n");
  return 0;
}
```

The main() function is always where your program starts running.

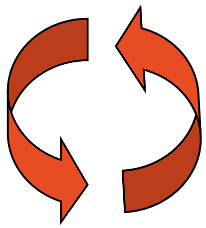
Blocks of code ("lexical scopes") are marked by { … }

Return '0' from this function

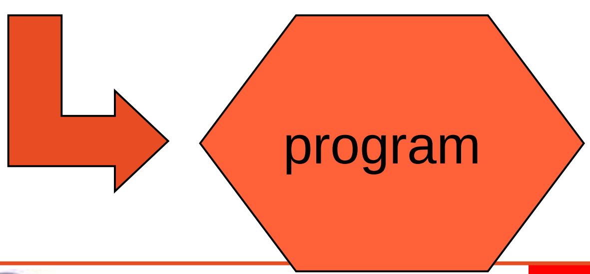
Print out a message. '\n' means "new line".

Writing and Running Programs

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```



```
$ cc program name.c
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and separately
tt.c:8: `x' undeclared (first use in this function)
tt.c:8: (Each undeclared identifier is reported only once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-void function
tt.c: At top level:
tt.c:11: parse error before `return'
```



1. Write text of program (source code) using an editor such as vi, gedit, save as file e.g. programname.c

2. Run the compiler to convert program from source to an "executable" or "binary":

   $ cc programname.c

3. Compiler gives errors and warnings; edit source file, fix it, and re-compile

4. Run it and see if it works ☺

   $ ./a.out
   Hello World
   $ ▌

OK, We're Back.. What is a Function?

A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.

"main()" is a Function. It's only special because it always gets called first when you run your program.

Return type, or void

Function Arguments

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Calling a Function: "printf()" is just another function, like main(). It's defined for you in a "library", a collection of functions you can call from your program.

Returning a value

```
/*———————————— PROGRAM USING FUNCTION ————————————*/
int mul (int a, int b); /*——— DECLARATION ————————*/
/*————————————— MAIN PROGRAM BEGINS —————————————*/
      main ()
      {
             int a, b, c;

             a = 5;
             b = 10;
             c = mul (a,b);

             printf ("multiplication of %d and %d is %d",a,b,c);
      }
/* ——————————————         MAIN PROGRAM ENDS
                          MUL() FUNCTION STARTS ——————————————————*/
         int mul (int x, int y)
         int p;
         {
                  p = x*y;
                  return(p);
         }
/* ———————————————————— MUL () FUNCTION ENDS ——————————————————*/
```

Fig. 1.7 *A program using a user-defined function*

EXECUTING A 'C' PROGRAM

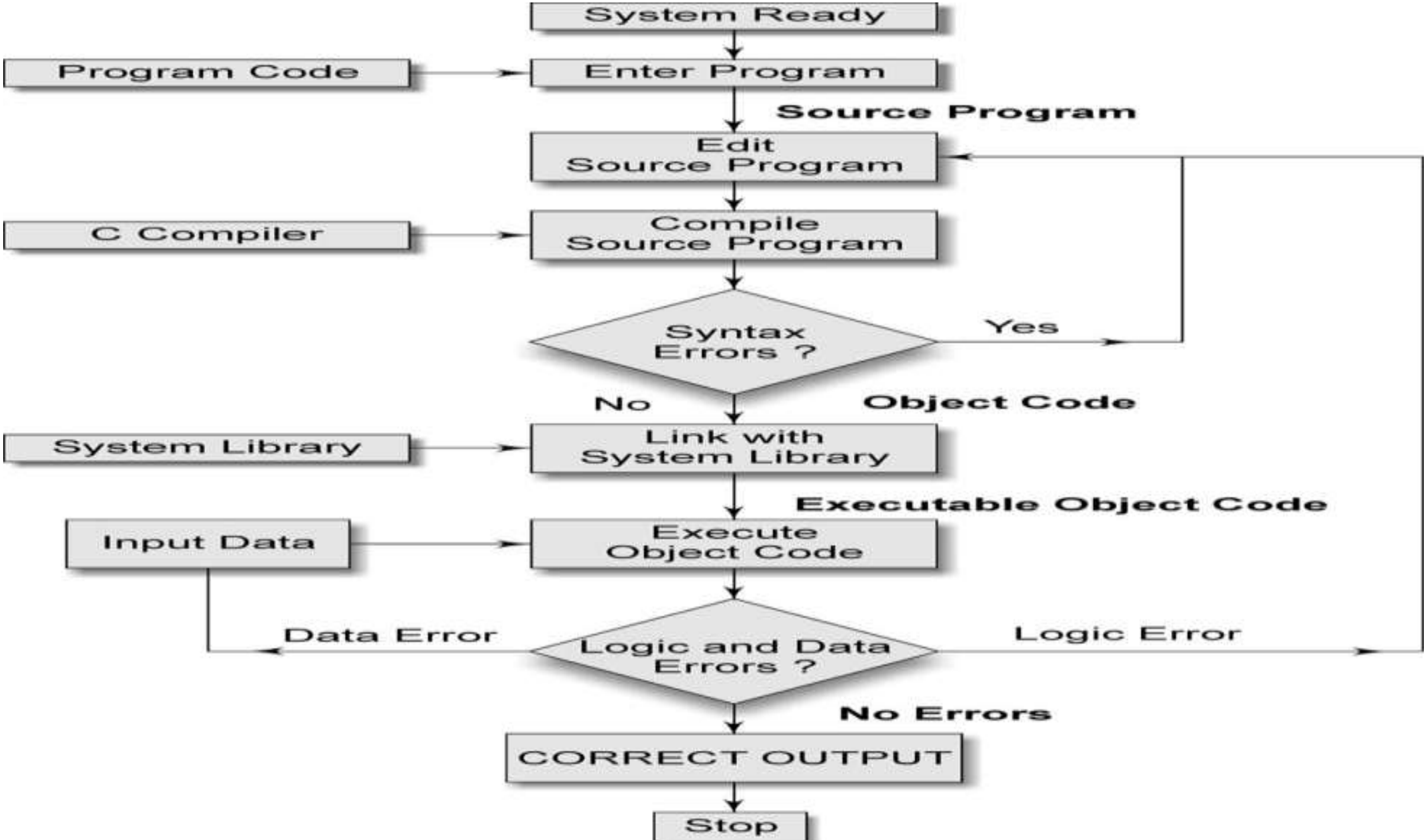


Fig. 1.10 Process of compiling and running a C program

Queries?

Module-2
Managing input/output
Decision Making & Branching

Swetha M S
Dept of ISE
BMSIT&M

Looping statements

- **Loop** is a control structure that repeats a group of steps in a program.
  - **Loop body** stands for the repeated statements.
- There are three C loop control statements:
  - **while**, **do-while and for**.

Comparison of Loop Choices (1/2)

Kind	When to Use	C Structure
Counting loop	We know how many loop repetitions will be needed in advance.	while, for
Sentinel-controlled loop	Input of a list of data ended by a special value	while, for
End file-controlled loop	Input of a list of data from a data file	while, for

Comparison of Loop Choices (2/2)

Kind	When to Use	C Structure
Input validation loop	Repeated interactive input of a value until a desired value is entered.	do-while
General conditional loop	Repeated processing of data until a desired condition is met.	while, for

The `while` Statement in C

- The syntax of **`while`** statement in C:

```
while (loop repetition condition)
{          statements;   }
```

- **Loop repetition condition** is the condition which controls the loop.
- The ***statement*** is repeated as long as the loop repetition condition is **true**.
- A loop is called an **infinite loop** if the loop repetition condition is always true.

Flow diagram –while loop

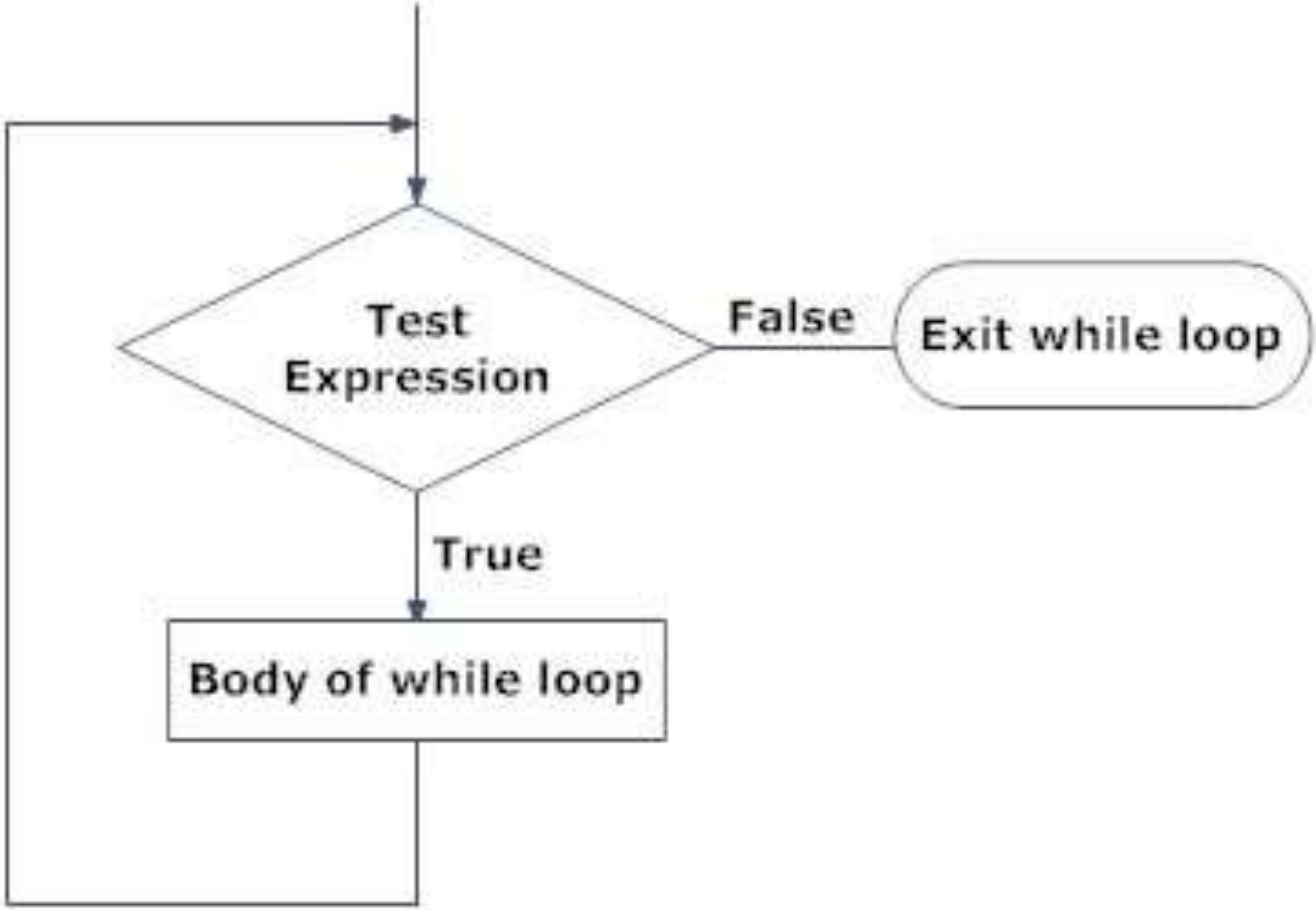


Figure: Flowchart of while loop

Check the given number is palindrome or not

```
int main( )
{           int n,temp,digit,rev=0;
            printf("enter a integer number\n");
            scanf("%d",&n);
            temp=n;
            while (n!=0)
            {           digit=n%10;
                        n=n/10;
                        rev=digit+10*rev;
            }     // while ends
            if(temp==rev)
            {
                        printf("%d is a PALINDROME\n",temp);
            }     // if ends
            else
            {                       printf("%d is not a PALINDROME\n",temp);          }
            return 0;
}
```

WAP to check whether the given number is armstrong or not
def: if the given number is equal to the sum of the cubes of individual digits then it is known as armstrong   ex: 407,153= 1 + 125 + 27 = 153

```
#include<stdio.h>
int main()
{
int r,s=0,n,m;
printf("Enter a number");
scanf("%d",&n);
m=n;
while(n>0)
{
r=n%10;
n=n/10;
s=s+(r*r*r);
}
if (m==s)
printf("The number is Armstrong");
else
printf("The number is not a Armstrong");
getch();
}
```

The `do-while` Statement in C

- The syntax of `do-while` statement in C:

```
do
    statement
while (loop repetition condition);
```

- The *statement* is first executed.
- If the **loop repetition condition** is true, the *statement* is repeated.
- Otherwise, the loop is exited.

Flow diagram do-while loop

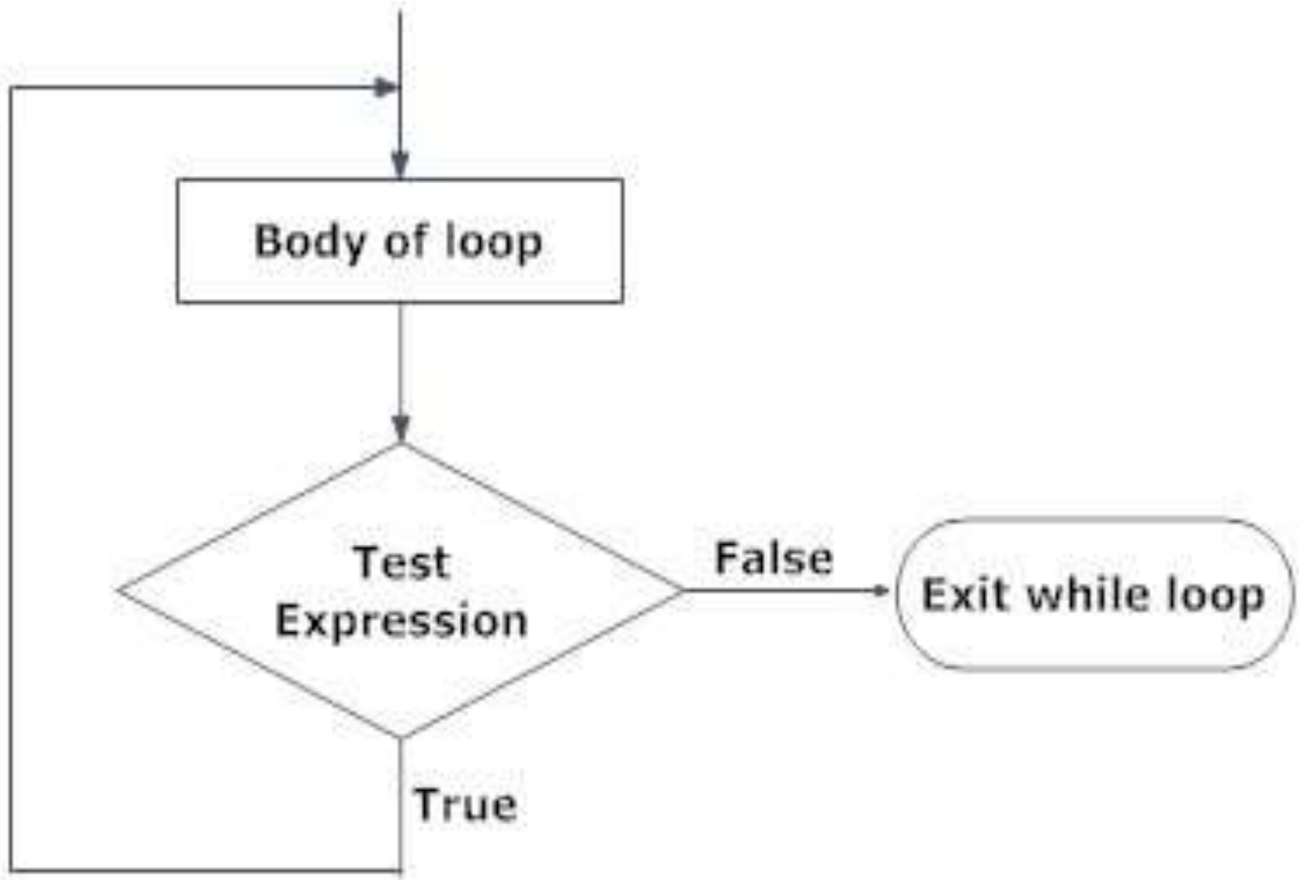


Figure: Flowchart of do...while loop

Example –do while (input validation loop)

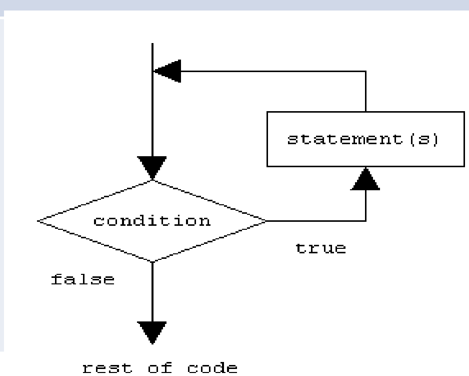
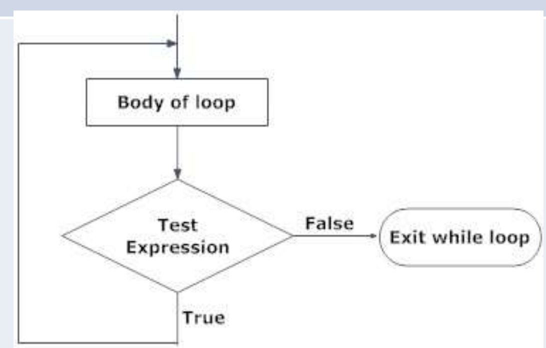
```
#include <stdio.h>
 int main( )
{ int sum=0,num;
do
{
 printf("Enter a number\n");
 scanf("%d",&num);
  sum+=num;
 } while(num!=0);
 printf("sum=%d",sum);
return 0;
}
```

An Example of the `do-while` Loop

```
/* Find even number input */
do{
   printf("Enter a value: ");
   scanf("%d", &num);
}while (num % 2 !=0)
```

This loop will repeat if the user inputs odd number.

Difference b/w while & do while

While	**Do while**
1. Entry controlled loop	1. Exit Controlled loop
2. If the condition is **FALSE ,** while loop is never executed.	2. If the condition is **FALSE** also in do-while loop, at least once statements can be executed.
3. syntax: while(condn) { stmts;}	**3. Syntax** do { stmts; }while(condn);
4. Flow diagram 	4. Flow Diagram  Figure: Flowchart of do...while loop

The `for` Statement in C

- The syntax of **`for`** statement in C:

`for` (**initialization** **;** **condition** **;** **update expr**)

{ *statements;*

}

- The **initialization expression** set the initial value of the loop control variable(using assignment operator =).
- The **condition** test the value of the loop control variable(using Relational Operator).
- The **update expression** update the loop control variable(using incr/decr operator).

Flow diagram -for loop

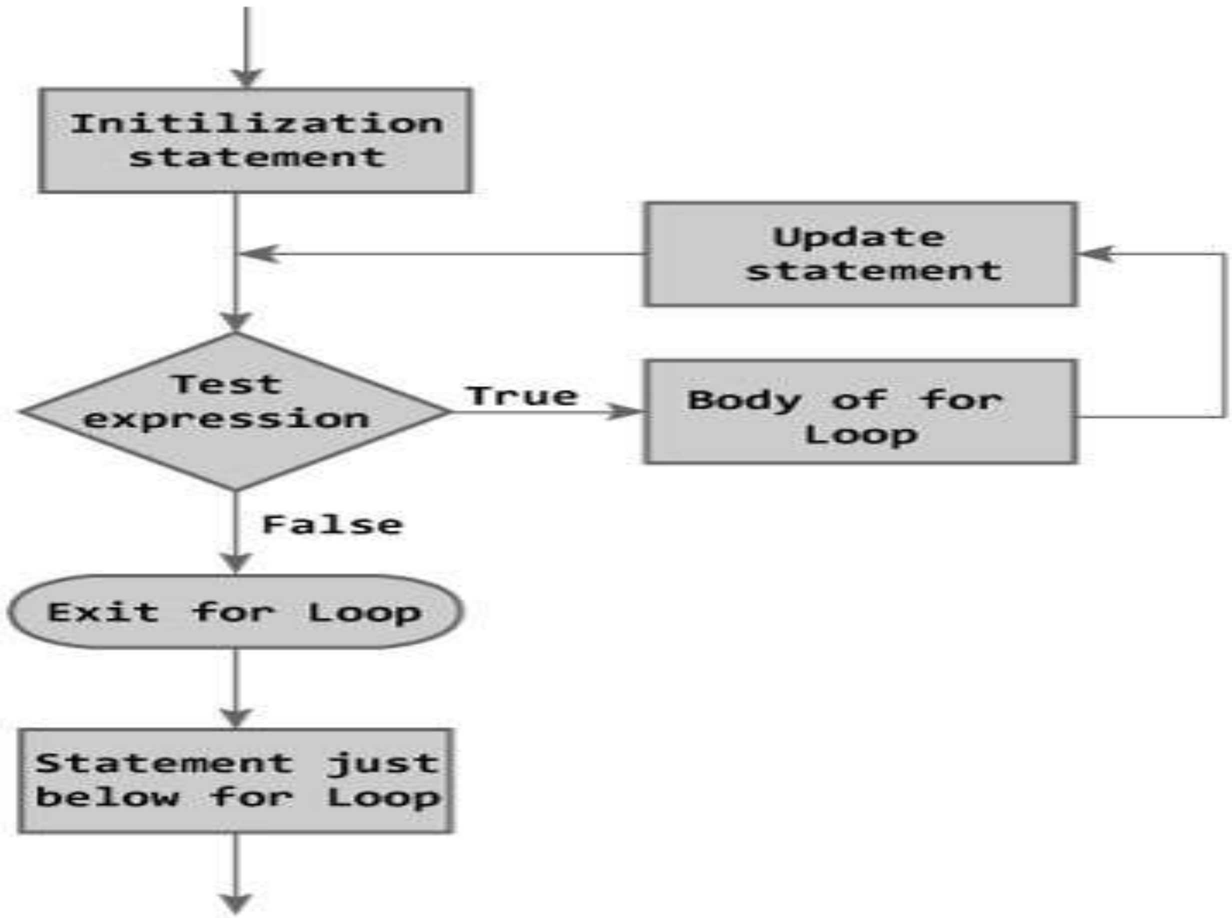


Figure: Flowchart of for Loop

for statement

```
for(i=0;i<=4;i++)
{
printf("\t%d",i)
}
printf("\n for loop over");
```

<u>O/P</u>

1	2	3	4	5

for loop over

Comparison of 3 loops

For loop	while	Do while

for (n=1;n<=10;n++)
{
    …………..
    …………..
}
	n=1;
	while(n<=10)
	{
	    …………..
	    …………..
	n=n+1
	}
		n=1;
		do
		{
		    …………..
		    …………..
		n=n+1;
		}
		while(n<=10);

C pgm to sum of first n natural numbers

```
#include <stdio.h>
int main()
{ int n, count, sum=0;
printf("Enter the value of
n.\n"); scanf("%d",&n);
for(count=1;count<=n;++count)
 { sum+=count; }
printf("Sum=%d",sum);
return 0; }
```

Output

Enter the value of n.
10

Sum=55

C pgm to find factorial of a given number

```
#include<stdio.h>
int main( )
{
 int i,f=1,num;
 printf("Enter a number: ");
 scanf("%d",&num);
 for(i=1;i<=num;i++)
     f=f*i;
 printf("Factorial of %d is: %d",num,f);
 return 0;
}
```

Additional Features of for

- More than1 variable can be initialized
- More than 1 part for update expression section
- Test condition can have any compound relation
- More than 1 parts in each section will be seperated by commas.

```
for(i=1,m=50; i<20 && sum <100; i++,m--)
```

Additional Features of for

- We can also use expression in the initialization and incr/decr part.

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

- One or more sections can be omitted, if necessary

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

Nested Loops

- Nested loops consist of an **outer loop** with one or more **inner loops**.
- e.g.,

```
for (i=1;i<=100;i++){
    for(j=1;j<=50;j++){
        …
    }
}
```

Outer loop

Inner loop

- The above loop will run for 100*50 iterations.

Eg pgm using nested for loops

```
#include <stdio.h>
int main()
{
int I,j;
for(i=1;i<=3;i++)
  {for(j=1;j<=3;j++)
   printf("i=%d    \t   j=%d", i,j);
}return 0;
}
```

O/P

i=1	j=1
i=1	j=2
i=1	j=3
i=2	j=1
i=2	j=2
i=2	j=3
i=3	j=1
i=3	j=2
i=3	j=3

C program to print Floyd's triangle:

```
#include <stdio.h>
 int main()
{
   int n, i, c, a = 1;
   printf("Enter the number of rows of Floyd's triangle
              to print\n");
   scanf("%d", &n);
    for (i = 1; i <= n; i++)
   {
     for (c = 1; c <= i; c++)
     {
        printf("%d ",a);
        a++;
     }
       printf("\n");
    }
return 0;
 }
```

Homework #4 (1/2)

- Write a program that prompts the user to input an integer ***n***.
- Draw a triangle with ***n*** levels by star symbols. For example,

  n = 3,

  *

  **

  ***

- After drawing the triangle, repeat the above process until the user input a negative integer.

Homework #4 (2/2)

- An usage scenario:

Please input: 2
*
**
Please input: 3
*
**
***
Please input: -9
Thank you for using this program.

Unconditional control Transfer)

- C permits a jump from one statement to another within a loop as well as the jump out of a loop.
- 4 unconditional control statements are available in C
  - goto (branching statement)
  - break (looping)
  - continue (looping)
  - return (used only in functions)

UNCONDITIONAL BRANCHING STATEMENTS (goto)

- C supports the goto statement to branch unconditionally from one point of the program to another.
- The goto requires a *label* in order to identify the place where the branch is to be made.
- A **label** is any valid variable name and must be followed by a colon.
- The general form is

```
goto label;
  ...
label:
```

UNCONDITIONAL BRANCHING STATEMENTS (goto)

```
goto label
----------
----------
label:
statement;
```

```
label:
statement;
----------
-----------
goto label
```

Note: Don’t use 2 labels with same name

Example for goto

```
#include<stdio.h>
#include<math.h>
main()
{
double x, y;
read:
printf(“Enter a No:”);
scanf(“%f”,&x);
if(x < 0)
goto read;
y = sqrt(x);
printf(“sqrt root of %f is %f \n”,x, y);
return 0;
}
```

Example-2

// pgm to print n natural numbers

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i=1;
clrscr();
printf("Enter the final value");
scanf("%d",&n);
printf("The natural numbers
are \n");
natural:
if(i<=n)
{
printf("%5d",i); i++;
goto natural;
}
getch();
}
```

Break statements

- The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done.**
- it is used to move the control outside of the control statements.
- **Syntax** break;

```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

NOTE: The break statment may also be used inside body of else statement.

Eg of break statment

```
main()
{
int t ;
for ( ; ; )   // infinite loop
{
printf(“\nEnter a Value:”);
scanf("%d" , &t) ;
if ( t==10 )
 break ;
}
printf("End of an infinite loop...\n");
}
```

NOTE: When the loops are nested, the **break** would only exit from the loop containing it.

That is, the **break** will exit only a single loop.

```
for(.......)
{
  .....
  for(.....)
{
if(condn)
break;
}
....
}
```

Exit from inner loop

4. Evaluate polynomial using Horner's method (LAB pgm) $f(x)=a_4x^4+a_3x^3+a_2x^2+a_1x+a_0$

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,sum,a[10],x;
sum=0;
printf("\nEnter the noof coefficients:");
scanf("%d",&n);
printf("Enter n+1 co-efficients: \n");
for(i=n;i>=0;i--)
{printf("\na[%d]=",i);
scanf("%d",&a[i]);
}
printf("\nEnter the Value of x:");
scanf("%d",&x);
for(i=n;i>=0;i--)
{
sum=sum*x+a[i];
}
printf("Sum is %d",sum);
getch();
}
```

continue

Syntax: continue;

- The keyword continue allows us to take the control to the beginning of the loop bypassing the statements inside the loop which have not yet been executed.

Pgm to show how continue works

```
#include<stdio.h>
main( )
{
int i,j;
for(i = 1; i< = 2; i++)
{
for(j=1; j<=2; j++)
{
if (i= =j)
continue;
printf("\n%d\t%d\n", i,j);
}
}
return 0
}
```

Output

12

21

Sum of positive elements

```
#include<stdio.h>
int main()
{
int a[5]={-1,2,-3,4,-5};
int i,sum=0;
for(i=0;i<5;i++)
{
if(a[i]<0)
  continue;
sum+=a[i];
}
printf("sum of positive elements: %d\n", sum);
return 0;
}
```

Difference b/w break & continue

S.No	Break	Continue
1.	Appears both in switch and loop(for, while ,do) statements	Appears only in loop(for,while,do) statements
2.	Used to exit from the loop immediately skipping one or more statements in the loop	Used to continue the loop, skipping one or more statements in the loop.
3.	Syntax: break;	Syntax: continue;

Module-3
Arrays

By
Prof Swetha M S
ISE-BMSIT&M

Syllabus-Arrays

- Using an array
- Using arrays with Functions
- Multi-Dimensional arrays

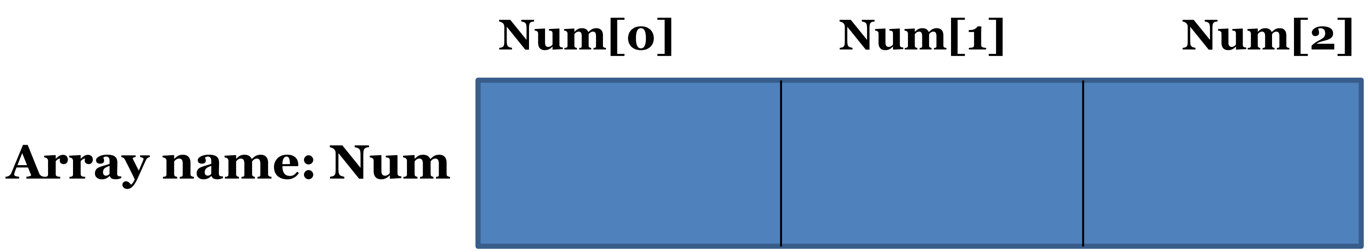
Arrays

Defn:

Array is a data structure that represents a collection of elements of same data type. (derived data type)

Syntax: datatype array_name[subsript/index/size];

Eg: int Num[3];



Need

- Easy to process large amount of data

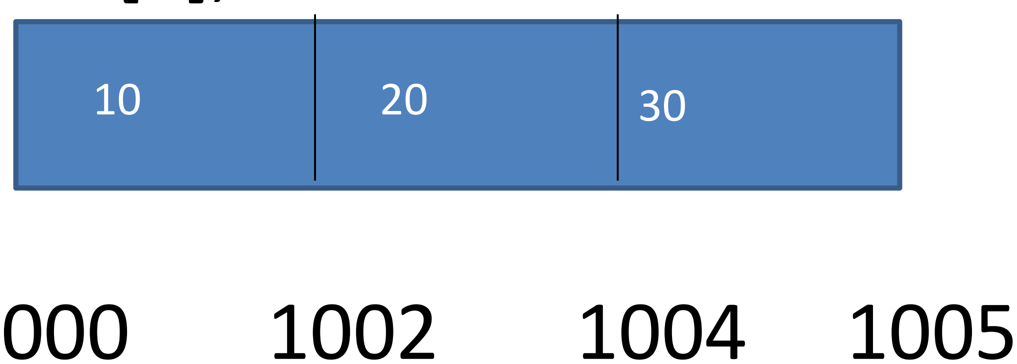
Classification of Arrays

- Single (one) Dimensional
- Two dimensional
- Multidimensional

Single dimensional Array

- Linear list consist of data items of same type.
- In memory all data items stored in continuous memory location.

eg. int a[3];



Properties of Array

1. Elements stored in array should be of same type
2. Elements are stored contiguously in memory.
3. Subscript of first item is always zero(if not specified)
4. Each data item is accessed using the name of the array
5. Index of the array is always an integer

    eg. a[2] √ correct

    a[2.5] X wrong

    a['5'] √ correct it takes ASCII value of '5'

    a[5+2] √ correct

Declaration of one dimensional Array

Syntax: Datatype array_name[index];

- int marks[5]; mem→2* 5=10 bytes
- float avg [3]; mem→4*3=12 bytes
- char name[5]; mem→1*5=5 bytes

Declaration using Named constants

```
const int SIZE=5;
int a[SIZE];
```

Declaration using Symbolic Constants

```
#define SIZE 3;
int marks[2+SIZE];
```

Storing values in Arrays

- Initialization
- Assigning values
- User input from keyboard

Initialization

data type array_name[index]={v1,v2,…vn};

eg→ int a[5]= {10,20,30,40,50};

Initialization

1. int a[3]={1,2,3,4,5} // Error number of values more than the size of the array
2. char name[5]={'K', 'U', 'M', 'A','R'};
3. int a[5]={10,20};
4. int a[ ]={10,20,30,40};

Array initialization with String

String is defined as sequence of character enclosed within double quotes ends with NULL(\0) character.

Char b[ ]= "WELCOME" // array size is equal to size of string +1

W	E	L	C	O	M	E	\0

- Note: Size of the array must be known during compilation.
- Eg:

```
  main()
 {
  int a[ ];  // error
  a[1]=20;
  …
  }
```

User input for arrays

- Using loops

Reading array Input

```
for(i=0;i<=n-1;i++)
 {
   scanf(“%d”,&a[i]);
}
```

Displaying ouput array

```
for(i=0;i<=n-1;i++)
{
printf(“%d”,a[i]);
}
```

Bubble sort

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,j,a[10],temp;
clrscr();
printf("Enter the No. of Elements:\n");
scanf("%d",&n);
printf(" Enter the array Elements:\n");
for(i=0;i<n;i++)
{
printf("\t");
scanf("%d",&a[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1+i;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
{
printf("\t");
printf("%d",a[i]);
}
getch();
}
```

Output

Enter the No. of Elements: 5

Enter the array Elements:

5 3 1 2 4

The sorted elements are

1 2 3 4 5

Copy one array to another array

```
#include<stdio.h>
int main() {
  int arr1[30], arr2[30], i, num;
   printf("\nEnter no of elements:");
  scanf("%d", &num);
   //Accepting values into Array
  printf("\nEnter the values :");
  for (i = 0; i < num; i++)
 {
     scanf("%d", &arr1[i]);
  }
// Copying data from array 'a' to array 'b '
  for (i = 0; i < num; i++)
{
    arr2[i] = arr1[i];
  }
   //Printing of all elements of array
  printf("The copied array is :");
  for (i = 0; i < num; i++)
{
 printf("\narr2[%d] = %d", i,arr2[i]);
 }
  return (0);
}
```

Output

Enter no of elements : 5

Enter the values : 11 22 33 44 55

The copied array is : 11 22 33 44 55

H/w

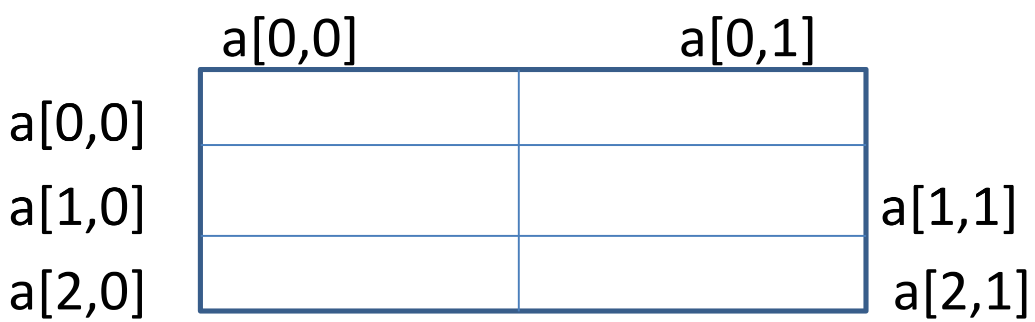
- average of array elements
- Find max value in an array
- Sum of odd and even numbers in an array

Two dimensional arrays

- A two dimensional array stores data as a logical collection of **rows** and **columns**.
- Also called arrays of arrays (matrix)
- Each element of a two-dimensional array has a row position and a column position.
- Syntax:
  - data type array_name[row][col];
  - eg: int array[5][3];

Creating/declaring 2-D arrays

int a[3][2];



- Row size and col size must be integers.

Initializing 2D arrays

- `int data[2][5];` //allocates consecutive memory for 10 integer values

Initialized directly in the declaration statement

- `double t[2][2] = {{3.0,5.0},{2.1,7.2}};` //allocates and initializes

(Or)

`t[0][0]= 3.0; t[0][1]=5.0; t[1][0]=2.1; t[1][1]=7.2`

3.0	5.0	2.1	7.2

1 st row   2nd row

```
int c[  ][3] = { {1, 2, 3},
                 {4, 5, 6},
                 {7, 8, 9},
                 {10, 11, 12} };
```

Programming Error: Do not specify more values than the number of elements declared for the array.

Input of Two-Dimensional Arrays

- Data may be input into two-dimensional arrays using nested *for* loops interactively or with data files.

```
for (i = 0; i < 2; i++)
  {
     for(j = 0; j < 3; j++)
     {
      scanf(“%d “, &a[i][j]);
      }
     printf(“\n”);
  }
```

Output of Two-Dimensional Arrays

- Nested *for* loops are used to print the rows and columns in row and column order.

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

```
for (i = 0; i < 2; i++)
{
    for(j = 0; j < 3; j++)
   {
   printf(“%d      “, a[i][j]);
    }
    printf(“\n”);
}
```

Matrix addition

```
#include <stdio.h>
 int main()
{
int m, n, c, d, first[10][10], second[10][10], sum[10][10];
printf("Enter the number of rows and columns of matrix\n");
scanf("%d%d", &m, &n);
printf("Enter the elements of first matrix\n");
 for (c = 0; c < m; c++)
{
    for (d = 0; d < n; d++)
     {
        scanf("%d", &first[c][d]);
      }
}
```

```
printf("Enter the elements of second matrix\n");
for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
       scanf("%d", &second[c][d]);

printf("Sum of entered matrices:-\n");
 for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
     {
        sum[c][d] = first[c][d] + second[c][d];
//Sum[0][0]=first[0][0]+sec[0][0]
//Sum[0][1]
         printf("%d\t", sum[c][d]);
      }
     printf("\n");

 }
 return 0; }
```

E:\programmingsimplified.com\c\add-matrix.exe

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1 2
3 4
Enter the elements of second matrix
5 6
2 1
Sum of entered matrices:-
6       8
5       5
```

Matrix Multiplication

“For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix”

2x2   3x3

```
#include <stdio.h>
include<process.h>
 int main()
{
int r1, c1,r2,c2 i, j,k, a[5][5], b[5][5], c[5][5];
printf("Enter the number of rows and columns of matrix A\n");
scanf("%d%d", &r1, &c1);
printf("Enter the number of rows and columns of matrix B\n");
scanf("%d%d", &r2, &c2);
if(c1!=r2)
  {
   printf(“Matrix Multiplication not possible\n”);
   exit(0);
   }
```

```
else
{
printf("Enter the elements of first matrix\n");
 for (i = 1; i < =r1; i++)
     for (j = 1; j <= c1; j++)
             scanf("%d", &a[i][j]);

printf("Enter the elements of second matrix\n");
for (i = 1; i <=r2; i++)
     for (j = 1 ; j < =c2; j++)
        scanf("%d", &b[i][j]);
```

```
for(i=1;i<=r1;i++)
{
   for(j=1;j<=c2;j++)
   {
     c[i][j]=0;
      for(k=1;k<=c1;k++)
     {
     c[i][j]=c[i][j]+a[i][k]*b[k][j];
     }
   }
}
```

```
printf("The product of 2 Matrices are:\
for(i=1;i<=r1;i++)
{
for(j=1;j<=c2;j++)
  {
    printf("%d \t",c[i][j]);
   }
  printf("\n");
 }
}
return 0;
}
```

```
Enter the number of rows and columns of matrix A
2
4
Enter the number of rows and columns of matrix B
2
2
Matrix Multiplication not possible

Enter the number of rows and columns of matrix A
2
2
Enter the number of rows and columns of matrix B
2
2
Enter the elements of first matrix
1       1
2       2
Enter the elements of second matrix
1       1
2       1
The product of 2 Matrices are:
3               2
6               4
```

Module-3

Strings

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Strings

- A string is a array of characters enclosed with in double quotes terminated with a null character(\0).

W	E	L	C	O	M	E	\0

- The operations that are performed on character strings are
  - ❖ Reading and writing strings.
  - ❖ Combining strings together.
  - ❖ Copying one string to another.
  - ❖ Comparing strings for equality.
  - ❖ Extracting a portion of a string.

Declaration of Strings

Syntax

char<var name>[array_length];

Eg:

```
char msg[6];
char str1[10],str2[10],.....;
```

Initialization of Strings

Syntax

char str[size]=string constant;

char str[ ]=string constant;

Eg:

char msg[10]="Hello";

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

char msg [] = " Hello";

H	E	L	L	O	\0

char msg[ ]={'H','e','l','l','o','\0'};

H	E	L	L	O	\0

Reading and displaying strings

- Read a string→ scanf(), gets(), getchar()
- Display a string→ printf(), puts(), putchar()

i) Using scanf() and printf()

scanf("%s",&msg); → format specifier is %s

printf(" Hello");

printf("%s", msg);

read and display string using scanf and printf

```
#include<stdio.h>
int main()
{
char name[20];      // declaration of string;
printf("\nEnter your name:");
scanf("%s", &name);     // & is optional for string
printf("\nwelcome %s !!!\n", name);
return 0;
}
```

Output1:

Enter your name: BMSIT
Welcome BMSIT!!!

Output2:

Enter your name: BMSIT BANGALORE
Welcome BMSIT !!!

Read and display string using gets and puts function

```
#include<stdio.h>
#define size 20
int main()
{
char name[20];
printf("Enter the name with space:");
gets(name);
printf("\nyou are:");
puts(name);
return 0;
}
```

Output:

Enter the name with space: dennis ritchie

You are : dennis ritchie

Difference between scanf() and gets()

scanf()

1. Reads input till space, it omits characters after blank space.

2.Syntax

scanf("format specifier", str_var);

3. eg:

char str[10];

scanf("%s", str);

gets()

1. Reads input including space till enter key is pressed

2.Syntax

gets(str_var);

3. eg:

char str[10];

gets(str);

Read and display strings using getchar() and putchar()

```
#include<stdio.h>
#define size 20
int main()
{
char name[20];
int i=0;
printf("enter the name :");
while((name[i]=getchar())!='\n')
{
i++;
}
name[i]='\0';
printf("you are:");
for(i=0;name[i]!='\0';i++)
putchar(name[i]);
return 0;
}
```

Output:

enter the name: BMSIT

You are: BMSIT

Creating array of strings

```
#include<stdio.h>
#include<conio.h>
int main()
{
char days[7][10]={"Sunday","Monday",”Wednesday","Thursday","Friday","Saturday"};
for(i=0;i<7;i++)
{
printf("%s \t",days[i]);
}
getch();
return0;
}
```

String Manipulation functions

#include<string.h>

String manipulation functions:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strlen()

<u>Syntax:</u>

temp_variable = strlen(string_name);

Eg:

```
#include<stdio.h>
#include <string.h>
int main()
{
char a[20]="Program";
  int length;
 length=strlen(a);
 printf("Length of string a=%d \n",length);    //calculates the length of string before null
                                                    charcter.
return 0;
}
```

<u>Output:</u>

Length of string a=7

Calculate the length of the string without using string functions

```
#include <stdio.h>
int main()
{
char s[20];
int i;
printf("Enter a string: ");
scanf("%s",s);
for(i=0; s[i]!='\0'; ++i);
    printf("Length of string: %d",i);
return 0;
}
```

Output:

Enter a string: welcome

Length of string:7

strcmp function

Syntax:

temp_varaible=strcmp(string1,string2);

- It compares the two strings and returns an integer value.
- If both the strings are same (equal) then this function would return 0
- otherwise it may return a negative or positive value based on the comparison.

Eg for strcmp

```c
#include <stdio.h>
#include <string.h>
int main()
{
char str1[30],str2[30];
 printf("Enter first string: ");
 gets(str1);
 printf("Enter second string: ");
 gets(str2);
 if(strcmp(str1,str2)==0)
      printf("Both strings are equal");
else
     printf("Strings are unequal");
return 0;
 }
```

Output1

Enter first string: Apple
Enter second string: Apple
Both strings are equal.

Output2

Enter first string: Apple
Enter second string: cat
strings are unequal.

Compare two string without strcmp()

```
#include<stdio.h>
 int main() {
   char str1[30], str2[30];
    int i;
    printf("\nEnter two strings :");
   gets(str1);
   gets(str2);
    i = 0;
   while (str1[i] == str2[i] && str1[i] != '\0')
     i++;
   if (str1[i] > str2[i])
      printf("str1 > str2");
   else if (str1[i] < str2[i])
      printf("str1 < str2");
   else
     printf("str1 = str2");
    return (0);
}
```

Output1:

Enter two strings:

apple

apple

str1=str2

Output2:

Enter two strings:

apple

cat

str1<str2

strcat

- strcat() concatenates(joins) two strings.
- It takes two arguments, i.e, two strings and resultant string is stored in the first string specified in the argument.

Syntax

- strcat(first_string,second_string);

Pgm for strcat

```
#include <stdio.h>
#include <string.h>
int main()
{
char s1[10] = "Hello";
char s2[10] = "World";
strcat(s1,s2);
printf("Output string after concatenation: %s", s1);
 return 0;
}
```

Output:

Output string after concatenation: HelloWorld

Pgm to concatenate two strings without strcat

```
#include <stdio.h>
 int main()
{
char s1[10], s2[10], i, j;
printf("Enter first string: ");
scanf("%s",s1);
printf("Enter second string: ");
scanf("%s",s2);
for(i=0; s1[i]!='\0'; ++i);     /* i contains length of string s1. */
    for(j=0; s2[j]!='\0'; ++j, ++i)
        {  s1[i]=s2[j]; }
  s1[i]='\0';
 printf("After concatenation: %s",s1);
return 0;
}
```

Output:

Enter first string: hello

Enter second string: world

The concatenated string is: helloworld

Strcpy()

- Function strcpy() copies the content of one string to the content of another string.
- It takes two arguments.

Syntax

- strcpy(destination,source);

Pgm for strcpy

```
#include <stdio.h>
#include <string.h>
int main()
{
char a[10],b[10];
 printf("Enter string: ");
 gets(a);
  strcpy(b,a);   //Content of string a is copied to string b.
printf("Copied string: ");
puts(b);
return 0;
 }
```

Output:

Enter string: hai

Copied String: hai

Pgm to copy one string to another without strcpy

```
#include <stdio.h>
int main()
{
   char s1[10], s2[10], i;
   printf("Enter string s1: ");
   scanf("%s",s1);
   for(i=0; s1[i]!='\0'; ++i)
   {
     s2[i]=s1[i];
    }
s2[i]='\0';
printf("String s2: %s",s2);
 return 0;
 }
```

Output:

Enter String s1: hello

String s2: hello

Bubble sort

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,j,a[10],temp;
clrscr();
printf("Enter the No. of Elements:\n");
scanf(“%d”,&n);
printf(“ Enter the array Elements:\n”);
for(i=0;i<n;i++)
{
printf("\t");
scanf("%d",&a[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1+i;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf(“\nThe sorted elements are:\n");
for(i=0;i<n;i++)
{
printf("\t");
printf("%d",a[i]);
}
getch();
}
```

Module-3

Linear search, Binary Search Bubble sort and Selection sort

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Linear Search

```
#include<stdio.h>
int main()
{
  int array[10], search, c, n;
printf("Enter number of elements in array\n");
 scanf("%d", &n);
 printf("Enter %d integer(s)\n", n);
for (c = 0; c < n; c++)
   scanf("%d", &array[c]);
 printf("Enter a number to search\n");
  scanf("%d", &search);
 for (c = 0; c < n; c++)
 {
  if (array[c] == search)
/* If required element is found */
  {
 printf("%d is present at location %d.\n", search, c+1);
    break;
  }
 }
 if (c == n)
  printf("%d isn't present in the array.\n", search);
 return 0;
}
```

Output

Enter the No. of Elements: 4

Enter the array Elements:

11 5 77 32

Enter the elements to serach-77

Element found at position 3

Bubble sort

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,j,a[10],temp;
//clrscr();
printf("Enter the No. of Elements:\n");
scanf("%d",&n);
printf(" Enter the array Elements:\n");
for(i=0;i<n;i++)
{
printf("\t");
scanf("%d",&a[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
{
printf("\t");
printf("%d",a[i]);
}
getch();
}
```

Output

Enter the No. of Elements: 5

Enter the array Elements:

5 1 4 2 8

The sorted elements are

1 2 4 5 8

Selection sort

```
#include <stdio.h>
int main()
{
  int array[10], n, i, j, min, temp;
  printf("Enter number of
elements\n");
  scanf("%d", &n);
 printf("Enter %d integers\n", n);
 for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
for (i = 0; i < (n - 1); i++) // finding
minimum element (n-1) times
  {
    min = i;
   for (j = i + 1; j < n; j++)
{
      if (array[j]<array[min])
        min = j;
    }
}
 t = array[i];
      array[i] = array[min];
      array[min] = t;
    }
 }
printf("Sorted list in ascending
order:\n");
 for (i = 0; i< n; i++)
   printf("%d\n", array[i]);
 return 0;
}
```

Output

Enter the No. of Elements: 5

Enter the array Elements:

5 7 4 8 1

The sorted elements are

1 4 5 7 8

Module-4
Functions

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Functions

- Function is a sub program or a self contained block of statements used to perform a specific task.
- Every C program contains at least one function called main()
- **Types of functions**
  - Built in functions→ functions provided by C compiler eg: scanf(),printf(),sqrt(),sin()
  - User defined functions→ functions defined by user eg: main()

Advantages of user defined functions

- Provides modularity and thus reduces complexity of the program
- avoids repetition of code
- Easy to debug/test the program
- Reduces time and cost (functions created for one pgm can be reused to another pgm with little or no modification)

Function Definition

Syntax:

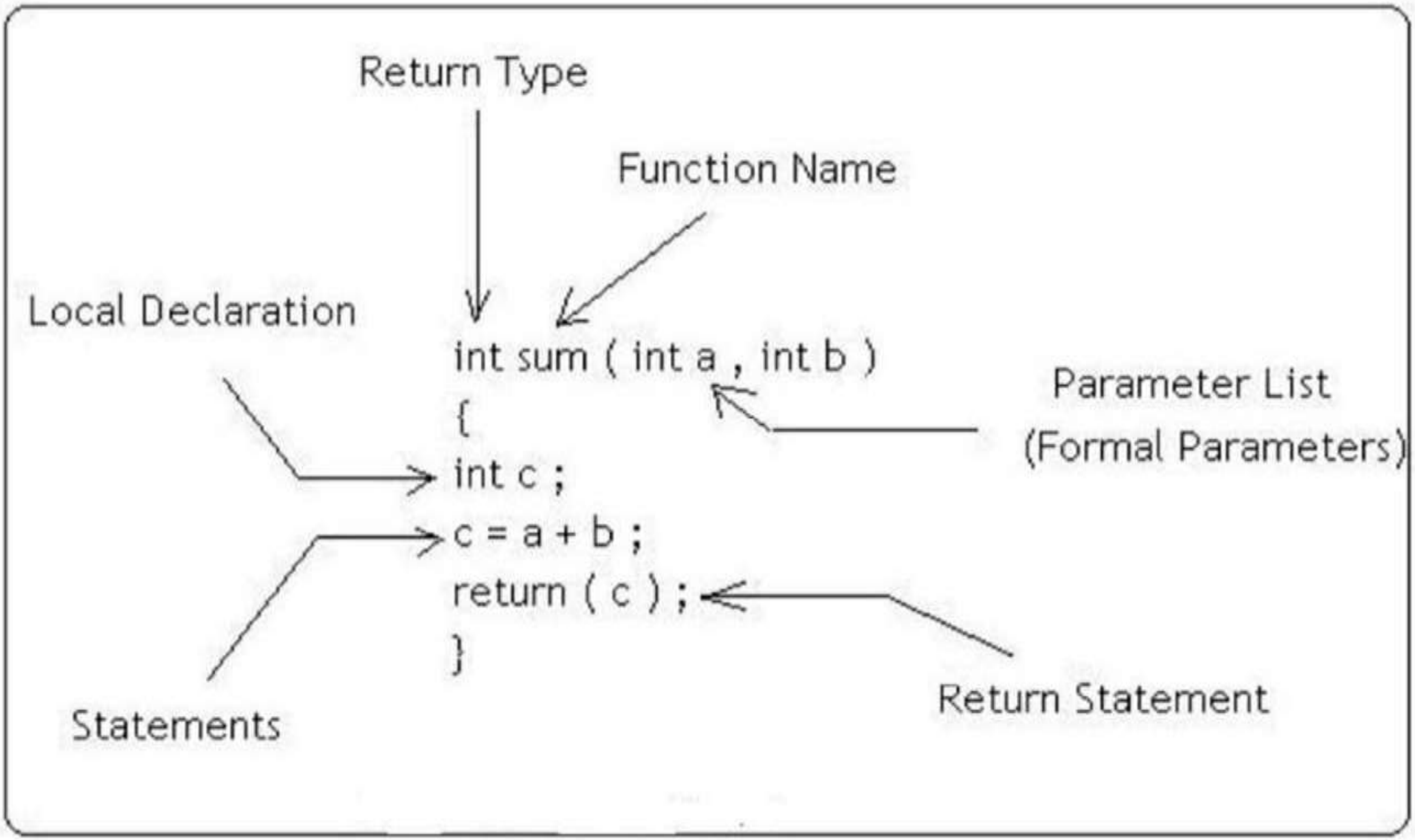
```
return_type function_name ( param list)
{
  declarations;
  executable_statments;
 }
```

Int add(int a, int b)

Note:

- A function can return only one value
- If the function returns no value then the return_type would be void.
- If the return_type is not specified by default type is int

Function Definition- Contd



Parameter list will take the following form

- type param1, type param2,....type param n
  - Eg: int add(int a, int b)
- Data type to be specified for each parameter
  - int max(int a b,c) - wrong
  - int max(int a, int b,int c) - correct

Examples-Program

Example: C program to add two numbers

```
/*without function*/
#include <stdio.h>
#include<conio.h>
void  main( )
 {

   int a,b,sum:-
   printf("enter a and b");
   scanf("%d%d", &a,&b);
   sum = a+b;
   printf("sum is %d", sum);
   getch( );
}
```

```
/*with user defined function*/
#include <stdio.h>
#include<conio.h>
int add(int a, int b);
void main( )
{
      int sum, a,b;
      printf("enter a and b");
      scanf("%d,%d",&a,&b);
      sum = add(a,b);
      printf("sum is %d",sum);
      getch( );
}
int add (int a, int b)
{
      int sum;
      sum=a+b;
      return sum;
}
```

Location of Functions

```
// after the main
Include header files
Function prototype
main()
{
}
Function definition()
{
}
```

```
// before the main
Include header files
Function prototype
Function definition()
{
}
main()
{
}
```

Location of Functions

Pgm1.c

Include header files

Function prototype

main()

{

}

Pgm2.c

Function definition()

{

}

Location of Functions-After the main

```
/*Documentation Section: program to find the sum of two integers*/
#include <stdio.h>/*link section*/

int addition( int , int ); /*Function Prototype declaration section*/

void   main()
{
        int  n1,n2,sum ;      /*declaration part*/

        printf("Enter the values of n1 and n2\n");   /*executable part starts
        here*/
        scanf("%d %d",&n1,&n2);
        sum = addition(n1,n2);/* Function Call */
        printf("Sum =%d \n ",sum);

}
int addition( int  x,int  y)   /* Function definition*/
 {
           int  s ;
           s = x + y;
           return  s;  /* Return statement to return the sum s*/
 }
```

Location of Functions-Before the main

```
int addition( int  x,int   y)   /* Function definition*/
 {
        int    s ;
        s = x+y;
        return s;   /* Return statement to return the sum s*/
  }
```

In the file say *addition.c* the following code has to be included

```
#include <stdio.h>/*link section*/

#include<add.h>/* the header file which includes the function
definition */
void    main()
{
        int    n1,n2,sum ;
        /*declaration part*/

        printf("Enter the values of n1  and n2\n");   /*executable part
        starts here*/
        scanf("%d %d",&n1,&n2);
        sum = addition(n1,n2); /* Function Call */
        printf("Sum =%d \n ",sum);

}
```

Actual and formal parameters

- Arguments/parameters appearing in function call is called actual arguments/parameters
- Arguments/parameters appearing in function definition is called formal arguments/parameters
- The number of actual and formal parameters must be equal.
- Also the data types and the order of declaration of formal and actual parameters must be the same.

Actual and formal parameters

Actual Parameters	Formal Parameters
1) Actual parameters are used in calling function when a function is invoked.	1) Formal parameters are used in the function header of a called function
2) Actual parameters can be constants, variables or expressions.	2) Formal parameters should be only variables.
3) Actual parameters send values to the formal parameters.	3) Formal parameters receive values from the actual parameters.
4) Address of actual parameters can be sent to formal parameters.	4) If formal parameters contains address, they should be declared as pointers.

- Each function will have
  - Function prototype
  - Function call
  - Function definition
  - Int add(int a, int b)
- If function is defined after main, function prototype is must .
- Function prototype tells the compiler which function is used by the main what is its return type , number and type of parameters.
- Name of parameters is optional in function prototype
- Function prototype should end with a semicolon

Program to Print a sentence using function

```
#include<stdio.h>
void display();                         //function declaration
void main()
{
display();                              //function call
}
void display()                          //function definition
{
printf("C Programming");
return;
}
```

```
#include<stdio.h>
int add(int , int );   // function prototype
int main()
{
int a=5,b=10;
add(a, b);       // function call
}
int add(int m, int n)     // function definition
{
int m,n,y;
Y= m+ n;
return(y);
}
```

Simple program using functions

```
#include<stdio.h>
int add(int,int);       // function prototype
int main()
{
int mark1=50,mark2=40,tot;
tot=add(mark1,mark2);    // function call mark1,mark2→ actual arguments
printf("the total is %d\n",tot);
return 0;
}
int add(int a,int b)   // function defintion a,b→ formal parameters
{
int c;
c=a+b;
return (c);     // returns the value of c to the variable tot
}
```

Note :

number of arguments, type of arguments, return type in function prototype should match with function defintion

```
int max( int a, int b); // fn prototype
 main()
{ ...
max(x,y);    // function call
 }
  int max( float a, float b, int c)  // fn defn
 {
 }    //This is wrong
```

Module-4
Functions

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Types of User-defined Functions in C Programming

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value.

Function with arguments and a return value.

Type1: with parameter with returning value

Parameters are passed from calling function to the called function and based on the receive parameter values called function performs required action and returns a value.

Example:

```
#include<stdio.h>
int add(int i, int j);
void main( )
{
        int sum, a=10, b=20;
        sum=add(a,b);
        printf("sum is %d", sum);
        getch( );
}
```

/*calling function main() passes the parameters a and b to the user defined function add( ). */

```
int add(int i, int j)
{
      int sum;
      sum = i+j;
      return sum;
}
```

/*here user defined function add( ) returning a value sum to calling function*/

Function type is int since it returns integer value to the called function.

Function with arguments and no return value

Type 2: with parameter without returning value:
Parameters are passed from calling function to the called function and called function does not return a value. It just performs the specified action.

Example:

```
#include<stdio.h>
void add(int i, int j);
void main( )
{
        int a=10, b=20;
        add(a,b);
        getch( );
}
```

/*calling function main() passes the parameters a and b to the user defined function add( ) and does not expect return value */

```
void add(int i, int j)
{
    int sum;
    sum = i+j;
    printf("sum is %d", sum);
}
```

/*here user defined function add( ) computes sum and itself display the value instead of returning to calling function*/

Function type is void since it returns nothing to the called function.

Function with no arguments and a return value

Type 3: without parameter with returning value:

No parameter is passed from calling function to the called function, but function returns a value.

Example:

```
#include<stdio.h>
int add( );
void main( )
{
        int sum;
        sum=add( );
        printf("sum is %d", sum);
        getch( );
}
int add( )
{
    int sum, i=10, j=20;
    sum = i+j;
    return sum;
}
```

/*calling function main() do not pass the parameters a and b to the user defined function add( ), but expect return value from the add( )*/

/*here the variables values (input) is given within the user defined function add( ) itself. Then computes sum and returns a value sum to calling function */

Function type is int since it returns integer value to the called function.

Function with no arguments and no return value

Type 4: without parameter without returning value:
No parameter is passed from calling function to the called function. Called function does not return any value.

Example:

```
#include<stdio.h>
void add( );
void main( )
{
        add( );
        getch( );

}
void add( )
{
      int sum, i=10, j=20;
      sum = i+j;
      printf("sum is %d", sum);
}
```

/*calling function main() do not pass the parameters to the user defined function add( ), and also do not expect return value from the add()*/

/*here the variables values (input) is given within the user defined function add( ) itself. Then computes sum and itself display the value instead of returning to calling function*/

Function type is void since it returns nothing to the called function.

Types of User-defined Functions in C Programming-example-2

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value.

No arguments passed and no return Value

Source Code of function having r

```
#include <stdio.h>

void checkPrimeNumber();

int main()
{
      checkPrimeNumber();     // no argument is passed to prime()
      return 0;
}

// return type of the function is void becuase no value is returned from t
void checkPrimeNumber()
{
      int n, i, flag=0;

      printf("Enter a positive integer: ");
      scanf("%d",&n);

      for(i=2; i <= n/2; ++i)
      {
            if(n%i == 0)
            {
                  flag = 1;
            }
      }
      if (flag == 1)
            printf("%d is not a prime number.", n);
      else
            printf("%d is a prime number.", n);
}
```

No arguments passed but a return value

```
{
      int n, i, flag = 0;

      // no argument is passed to the function
      // the value returned from the function is assigned to n
      n = getInteger();

      for(i=2; i<=n/2; ++i)
      {
            if(n%i==0){
                  flag = 1;
                  break;
            }
      }

      if (flag == 1)
            printf("%d is not a prime number.", n);
      else
            printf("%d is a prime number.", n);

      return 0;
}

// getInteger() function returns integer entered by the user
int getInteger()
{
      int n;

      printf("Enter a positive integer: ");
      scanf("%d",&n);

      return n;
}
```

Argument passed but no return value

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int
```

Source code of a function with arguments passed but no return value

```
{
      int n;

      printf("Enter a positive integer: ");
      scanf("%d",&n);

      // n is passed to the function
      checkPrimeAndDisplay(n);

      return 0;
}

// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n)
{
      int i, flag = 0;

      for(i=2; i <= n/2; ++i)
      {
            if(n%i == 0){
                  flag = 1;
                  break;
            }
      }
      if(flag == 1)
            printf("%d is not a prime number.",n);
      else
            printf("%d is a prime number.", n);
}
```

Argument passed and a return value

```
int main()
{
	int n, flag;

	printf("Enter a positive integer: ");
	scanf("%d",&n);

	// n is passed to the checkPrimeNumber() function
	// the value returned from the function is assigned to flag variabl
	flag = checkPrimeNumber(n);
                                                   Function wit argument and a retu
	if(flag==1)
		printf("%d is not a prime number",n);
	else
		printf("%d is a prime number",n);

	return 0;
}

// integer is returned from the function
int checkPrimeNumber(int n)
{
	/* Integer value is returned from function checkPrimeNumber() */
	int i;

	for(i=2; i <= n/2; ++i)
	{
		if(n%i == 0)
			return 1;
	}

	return 0;
}
```

Module-4
Functions

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Argument Passing

<u>**Parameter passing mechanism**</u>

- Call by value
- Call by reference

Call by value:

When the value of the variable is passed during function invocation is called call by value.

Call by reference:

When an address of the variable is passed during function invocation is called call by reference.

Call by value

```
main ( )
{
int a = 10, b=20;
swap(a,b);
printf ("\na = % d b = % d", a,b);
}
swap(int x, int y)
{
int  t;
t = x;
x = y;
y = t;
printf ( "\n x = % d y = % d" , x, y);
}
```

Output

x = 20 y = 10

a =10 b =20

Note:

With this method the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function

CALL BY REFERENCE

- The addresses of actual arguments in the calling function are copied into formal arguments of the called function.
- This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.
- Change in formal arguments affect the actual arguments

Call by reference

```
main ( )
{
int a = 10, b =20;
swap (&a, &b);
printf (“\n a = %d b= %d”, a, b);
}
swap (int *x, int * y)
{
int  t;
t = *x
*x = *y;
*y = t;
printf(“\n x=%d y=%d”,*x,*y);
}
```

<u>output</u>

x=20 y=10

a = 20 b =10

Passing entire array as arguments

```
#include <stdio.h>
float average(float a[]);         // fn prototype
int main(){
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
    avg=average(c);         /* Only name of array is passed as argument in fn call. */
    printf("Average age=%.2f",avg);
    return 0;
  }
float average(float a[])    // array var should be used as arg to receive the elements
{
    int i;
    float avg, sum=0.0;
    for(i=0;i<6;++i){
      sum+=a[i];
    }
    avg =(sum/6);
    return avg;
}
```

Passing arrays using Call by value

```
#include <stdio.h>
disp( char ch)          // display function definition
{
   printf("%c ", ch);
}
int main()
{
   char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'I', 'j'};
   for (int x=0; x<=10; x++)
   {
      /* passing each element one by one using subscript*/
      disp (arr[x]);          // fn call
   }

   return 0;
}
```

Passing array using call by reference

```
#include <stdio.h>
disp( int *num)
{
   printf("%d ", *num);
 }
 int main()
 {
     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
for (int i=0; i<=10; i++)
     {        /* passing element's address*/
             disp (&arr[i]);
      }
     return 0;
}
```

Sorting of elements using functions

```
#include<stdio.h>
void bubble_sort(int a[],int n)
{
            int i,j,temp;
            for(i=0;i<n-1;i++)
            {
              for(j=0;j<n-(i+1);j++)
               {
                   if(a[j]>a[j+1])
                    {
                       temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                     }
                 }
            }
}
```

```
int main()
{
   int a[10],i,n;
   printf("enter num of elements\n");
    scanf("%d",&n);
    printf("enter the elements of array\n");
     for(i=0;i<n;i++)
            scanf("%d",&a[i]);
       bubble_sort(a,n);
       printf("the sorted array is \n");
        for(i=0;i<n;i++)
              printf("%d\n",a[i]);
         return 0;
}
```

Passing Multidimensional array to functions

```
#include <stdio.h>
void Function(int c[2][2]);  // prototype
int main( )
{
  int c[2][2],i,j;
  printf("Enter 4 numbers:\n");
  for(i=0;i<2;++i)
    for(j=0;j<2;++j){
        scanf("%d",&c[i][j]);
    }
  Function(c);      // 2d array passed
  return 0;
}
```

```
void Function(int c[2][2])
{
/* Instead to above line, void
Function(int c[][2]){ is also valid */
  int i,j;
  printf("Displaying:\n");
  for(i=0;i<2;++i)
    for(j=0;j<2;++j)
        printf("%d\n",c[i][j]);
}
```

Module-4
Recursive Functions

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Recursion

- Recursive function is a function that calls itself from its own body.
- The function keeps on calling itself till a particular condition holds true.
- add(N+add(n-1))

Properties of Recursion

- There is a criteria or condition that governs the execution of recursive function. Without this condition, it will work in an endless manner. This condition is also called the base case of recursion.

Pgm to calculate factorial using recursion

```
factorial(5) = 5 * factorial(4)
             = 5 * 4*factorial(3)
             =5 * 4 *3*factorial(2)
             =5 * 4 * 3 *2*factorial(1)
             =5 * 4 * 3 * 2 *1*factorial(0)
             =5 * 4 * 3 * 2 * 1 * 1
             =120
```

That is factorial of n is n*factorial(n-1) where again factorial of n-1 is (n-1)*factorial(n-2) and so on until it reaches the stopping condition, factorial(0) =1. At each step factorial function calls itself and procedure is repeated until it reaches stopping condition. This concept is called **recursion in C programming**.

Pgm to calculate factorial using recursion

```
#include<stdio.h>
 int factorial (int );      //function prototyping
int main()
{   int num,result;
    printf("\nEnter a number : ");
    scanf("%d",&num);
    result= factorial(num);       // fn call
    printf("\nFactorial of  %d is %d",num,result);
  return 0;
}
```

Continue..

```
int factorial(int value)
{
    int ans;
    if( (value==0) || (value==1) )
      return(1);
    else
      ans = value* factorial(value-1);
        //call to itself
    return(ans);
}
```

Program for Sum of N numbers With Recursive call

```
#include<stdio.h>
int add(int n);
void main( )
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
   sum=add(n);
    printf("Sum = %d", sum);
    getch( );
}
int add(int n)
{
    if(n==0)
       return 0;
   else
                        return n+add(n-1);  /* recursive call */
}
```

Fibonacci series using Recursion

```
#include<stdio.h>
int fibbo(int x);
void main()
{
 int n,i;
 printf("Enter the number of
            terms in series\n");
scanf("%d",&n);
printf("Fibonacci series:\n");
for(i=1;i<=n;i++)
      printf("%ld\t",fibbo(i));
}
```

```
int fibbo(int x)
{ if(x==1 || x==0)
return 1;
 else
 {
return(fibbo(x-1)+ fibbo(x-2));
}
}
```

Void and parameter less functions

```
/*C program to check whether a number entered by user is
prime or not using function with no arguments and no return
value*/
#include <stdio.h>
void prime();     // fn prototype no parameter no return
int main()
{
prime();      //No argument is passed to prime()
return 0;
}
```

```
void prime()
{ /* There is no return value to calling function main(). Hence,
return type of prime() is void */
 int num,i,flag=0;
printf("Enter positive integer enter to check: \n");
scanf("%d",&num);
for(i=2;i<=num/2;++i)
{
if(num%i==0)
{
flag=1;
 }
}
if (flag==1)
printf("%d is not prime",num); else printf("%d is prime",num); }
```

Lab Program-12
Square Root of Given Number

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Lab Program-12

- ***Develop a program to find the square root of a given number N and execute for all possible inputs with appropriate messages.***
- ***Note: Don’t use library function sqrt(n).***

Program - 12

Finding the sqrt of a given number withou building fun

We need to know - for this program for loop

for (initialization ; condition ; incr/decr){ s1;
s2;
}

We are using Here 2 loops → S & D

S for decimal number

D - fractional number

Like example value is (n=50) - i/p

↓

7.071

decimal part ↓ S for loop

fractional part ↓ D - for Loop

S - for(s=1; s*s<=n; s++); → used to terminate for loop in $\boxed{s=8}$

S-- ✓ $\boxed{s=7}$

s=1;

1*1 <= 50

2*2 <= 50

3*3 <= 50

|

(49) 7*7 <= 50

(64) 8*8 <= 50 X condition terminates

which is more $\boxed{s=8}$ → more

So we have decrement of s--,

so, s value is 7 will be considered

D-Loop decimal

for (d=0.001; d<1.0; d+=0.001)

{

x=(double)S+d;

(o2-S is integer or decimal value, so doing typecasting for S

x=(double)7+0.071;

$\boxed{x=7.071}$

if(x*x > n)
{
x = x - 0.001;
break;
}
if((7.071)^2 > 50)

x = x - 0.001;

CO2 square value (fractional need to not exceed more)

Lab Program-15

Binary to Decimal Conversion

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

- **Purpose:** This program demonstrates **RECURSION**.
- **Procedure:** Input binary number and call the recursive function convert for translating binary number to decimal number.
- **Input:** A binary number *bin.*
- **Expected Output:** Decimal number

Implement Recursive functions for Binary to Decimal Conversion

```
#include <stdio.h>
int convert(int);
int main()
{
int dec, bin;
printf("Enter a binary number: ");
scanf("%d", &bin);
dec = convert(bin);
printf("The decimal equivalent of %d is %d.\n", bin,dec);
return 0;
}
```

Implement Recursive functions for Binary to Decimal Conversion

```
int convert(int bin)
{
if (bin == 0)
{
return 0;
}
else
{
return (bin % 10 + 2 * convert(bin / 10));
}
}
```

Implement Recursive functions

Binary to decimal

```
int convert (int bin)
{
  if(bin = = 0)
  {
    return 0;
  }
  else
  {
    return (bin % 10 + 2 * convert(bin)10 ));
  }
}
```

1001

$1001 \rightarrow 9$

$\cancel{1001\%10}$ return (binaryno % 10 +
2 * convert(bin / 10)

1) $(1001 \% 10) + 2 * \text{convert}(1001/10) \Rightarrow 100$ (convert(100) Recursive

$1 + 2 * [\, 100 \% 10 + 2 * \text{convert}(100/10) \,]$

$1 + 2 \times [\, 0 + 2 \times \text{convert}(10) \,]$

$1 + 2 * [\, 0 + 2 \times [\, 10 \% 10 + 2 * \text{convert}(10/10) \,]\,]$

$1 + 2 * [\, 0 + 2 * [\, 0 + 2 * (1) \,]\,]$

$1 + 2 * [\, 0 + 2 * [\, 0 + 2 \,]\,]$

$1 + 2 * [\, 0 + 2 * (2) \,]$

$1 + 2 * [4]$

$1 + 8$

$= 9$

<u>1010</u>

$(1010 \% 10) + 2*(1010/10)$

$0 + 2*((101 \% 10) + 2*(101/10))$

$0 + 2*(1 + 2*((10 \% 10) + 2(10/10)))$

$0 + 2*(1 + 2*(0) + 2(1))$

$0 + 2*(1 + 2*(2))$

$0 + 2*(1 + 4)$

$0 + 2*(5)$

$0 + 10$ decimal no is 10

Lab Program-9

Compute Sin(x) using Taylor series approximation

Prof. Swetha M S
Assistant Professor
ISE-BMSIT& M

Evaluating Sin(x) values without build in function and comparing result with build in function

- ***Develop a Program to compute Sin(x) using Taylor series approximation .Compare your result with the built- in Library function. Print both the results with appropriate messages***

Program

```
/* Program to calculate sine value of given angle */
#include<stdio.h>
#include<conio.h>
#include<math.h>
 #define PI 3.142
int main()
{
int i, degree;
float x, sum=0,term,nume,deno;
clrscr();
printf("Enter the value of degree");
scanf("%d",&degree);
x = degree * (PI/180); //converting degree into radian
nume = x;
deno = 1;
 i=2;
```

Program

```
do
{
//calculating the sine value.
term = nume/deno;
nume = -nume*x*x;
deno = deno*i*(i+1);
sum=sum+term;
 i=i+2;
} while (fabs(term) >= 0.00001); // Accurate to 4 digits
printf("The sine of %d is %.3f\n", degree, sum);
printf("The sine function of %d is %.3f", degree, sin(x));
return 0;
}
```

Program

(1)

Lab program – 9

Sine Series - Calculate the Sin(x) value without build-in fn

Taylor Series approximation:

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Numerter

Term

denominter

Program

Need to know about the do while statement

```
do
{
    Statement 1;
    Statement 2;
    .
    .
    Statement n;
}
while(condition);
```

Program

We need to convert degree
to
radian

A unit of measure of angles

X = degree * (PI/180);

3.14 → #define 3.142

nume = X;
demo = 1;

very first term

Program

```
do
{
    term = nume/deno;
    nume = -nume*x*x;
    deno = deno*i*(i+1);
    i=i+2
}while(fabs(term>0.000))
```

* you ppl know do-while executs the loop atleast once /* condition is checked ~~at~~ later */

each term is combination of & every term = num/deno

Program

3

& numo = - numo * x * x ;

(–) is for alternative + & – in the sin series

2-x is Every power is increasing by 2

So we have x*x → x^2

2 times

Program

denro = denro * i * (i+1)

Each & Every denro Need to be incremented by 2 & need to evaluate its factorial

& no more i = i+2 } cos of incremented by 2

while (fabs(term > 0.0000)); 0.00001

(Absolute value of floating point number)

check for maximum of 4 point accuracy

Module-5

C Pointers

Prof. Swetha M S

ISE-BMSIT&M

.

OBJECTIVES

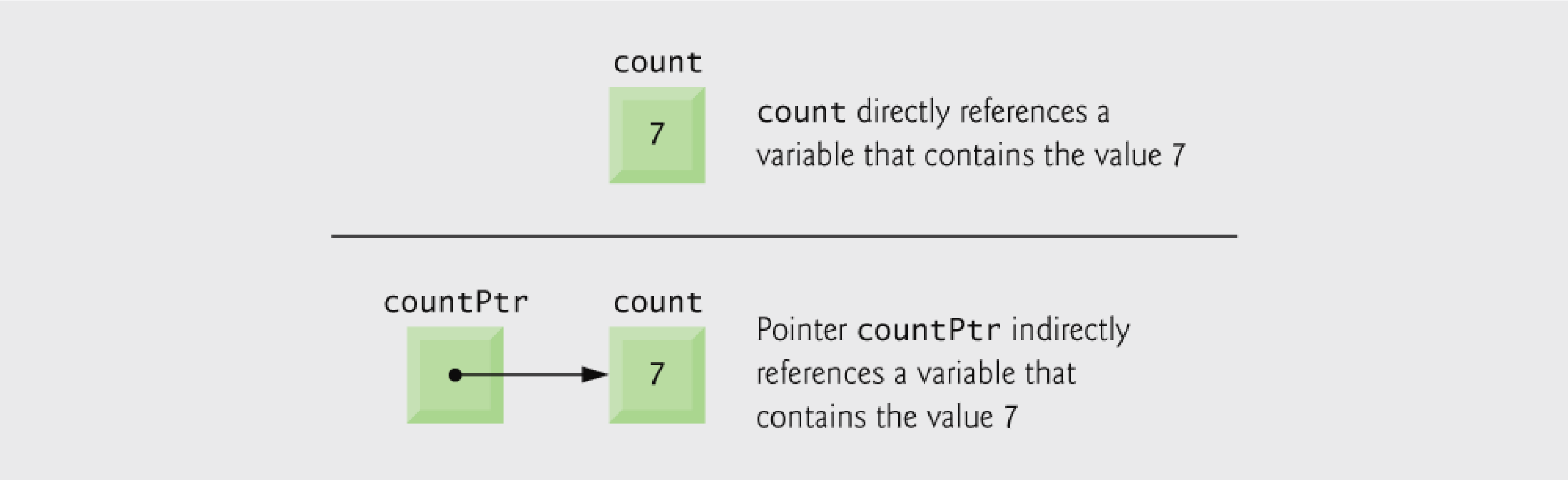
- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Character pointer and functions
- Pointer to pointer

Pointer Variable Definitions and Initialization

- Contain memory addresses as their values
- Normal variables contain a specific value (direct reference)

Defn

- A pointer is a variable which contains the address of a variable that has a specific value (indirect reference)



Pointer Variable Definitions and Initialization

- Pointer definitions
  - * used with pointer variables

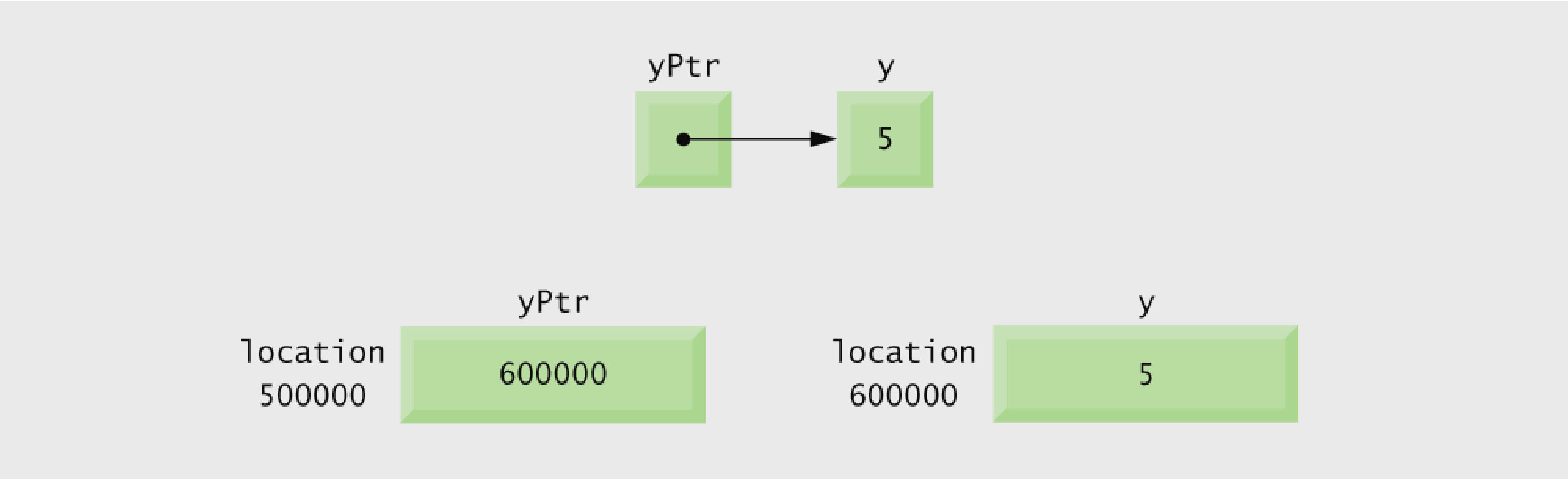
    ```
    int *myPtr;
    ```
  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a `*` before each variable definition

    ```
    int *myPtr1, *myPtr2;
    ```
  - Can define pointers to any data type
  - Initialize pointers to 0, `NULL`, or an address
    - 0 or `NULL` – points to nothing
    - 0 is the only integer value that can be assigned directly to a pointer variable.
    - Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred

Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;      /* yPtr gets address of y */
yPtr “points to” y
```



Pointer Operators

- `*` (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yptr` returns `y` (because `yptr` points to `y`)
  - `*` can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

  - Dereferenced pointer (operand of `*`) must be an lvalue (no constants)
- `*` and `&` are inverses
  - They cancel each other out

- fig07_04.c (1 of 2)

```c
/*Using the & and * operators */
#include <stdio.h>

int main( void )
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

If `aPtr` points to `a`, then `&a` and `aPtr` have the same value.

`a` and `*aPtr` have the same value

`&*aPtr` and `*&aPtr` have the same value

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

- fig07_04.c (2 of 2)

Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer
- * operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - *number used as nickname for the variable passed

```
/*Cube a variable using call-by-value */
        #include <stdio.h>
        int cubeByValue( int n );   /* prototype */
        int main( void )
        {
            int number = 5;
            printf( "The original value of number is %d", number );
          /* pass number by value to cubeByValue */
             number = cubeByValue( number );
              printf( "\nThe new value of number is %d\n", number );
              return 0;
        }
          int cubeByValue( int n )
          {
          return n * n * n;
          }
```

The original value of number is 5
The new value of number is 125

Call-by-reference with a pointer argument

```
#include <stdio.h>
void cubeByReference( int *nPtr ); /* prototype */
int main( void )
{
  int number = 5;
printf( "The original value of number is %d", number );
/* pass address of number to cubeByReference */
 cubeByReference( &number );
printf( "\nThe new value of number is %d\n", number );
 return 0;
 }
void cubeByReference( int *nPtr )
{
 *nPtr = *nPtr *x*nPtr x *nPtr;  /* cube *nPtr */
}
```

Function prototype takes a pointer argument

Function **`cubeByReference`** is passed an address, which can be the value of a pointer variable

In this program, **`*nPtr`** is **`number`**, so this statement modifies the value of **`number`** itself.

Analysis of A Typical Call-by-Value

Step 1: Before main calls cubeByValue:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number
5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n
undefined

Step 2: After cubeByValue receives the call:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number
5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n
5

Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number
5

```
int cubeByValue( int n )
{
               125
   return n * n * n;
}
```

n
5

Step 4: After cubeByValue returns to main and before assigning the result to number:

```
int main( void )
{
   int number = 5;
                   125
   number = cubeByValue( number );
}
```

number
5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n
undefined

Step 5: After main completes the assignment to number:

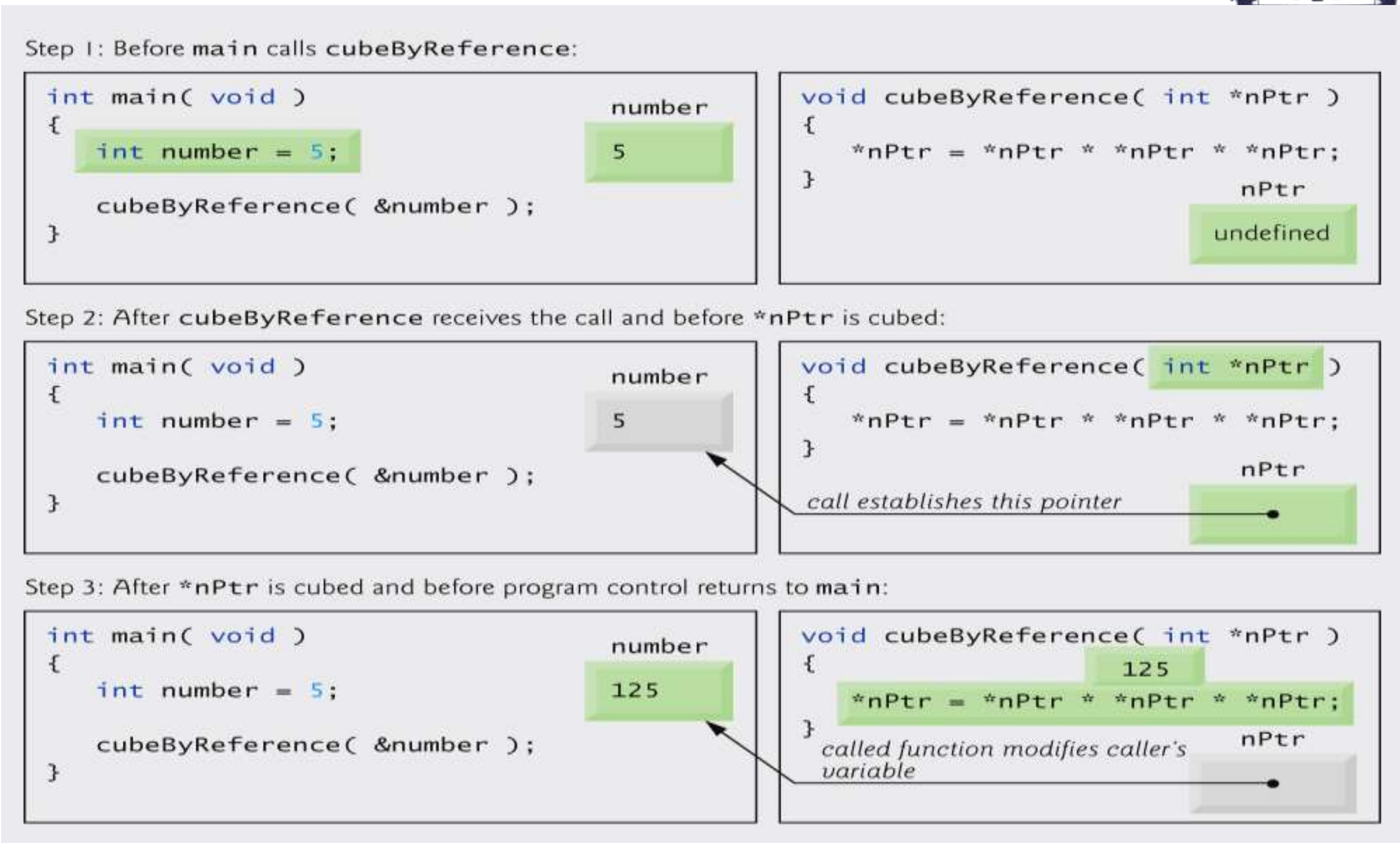
```
int main( void )
{
   int number = 5;
     125              125
   number = cubeByValue( number );
}
```

number
125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n
undefined

Analysis of A Typical Call-by-Reference



Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - Swap two elements
  - swap function must receive address (using &) of array elements
    - Array elements have call-by-value default
  - Using pointers and the * operator, swap can switch array elements
- Psuedocode

    Initialize array

    print data in original order

    Call function bubblesort

Bubble Sort Using Call-by-reference

- Psuedocode

    Initialize array

    print data in original order

    Call function bubblesort

    print sorted array

    Define bubblesort

```c
/* Fig. 7.15: fig07_15.c
   This program puts values into an array, sorts the values into
   ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int * const array, const int size ); /* prototype */

int main( void )
{
   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   int i; /* counter */

   printf( "Data items in original order\n" );

   /* loop through array a */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */

   bubbleSort( a, SIZE ); /* sort the array */

   printf( "\nData items in ascending order\n" );

   /* loop through array a */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */
```

- fig07_15.c (1 of 3)

```c

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* sort an array of integers using bubble sort algorithm */
void bubbleSort( int * const array, const int size )
{
   void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
   int pass; /* pass counter */
   int j;    /* comparison counter */

   /* loop to control passes */
   for ( pass = 0; pass < size - 1; pass++ ) {

      /* loop to control comparisons during each pass */
      for ( j = 0; j < size - 1; j++ ) {

         /* swap adjacent elements if they are out of order */
         if ( array[ j ] > array[ j + 1 ] ) {
            swap( &array[ j ], &array[ j + 1 ] );
         } /* end if */

      } /* end inner for */

   } /* end outer for */

} /* end function bubbleSort */
```

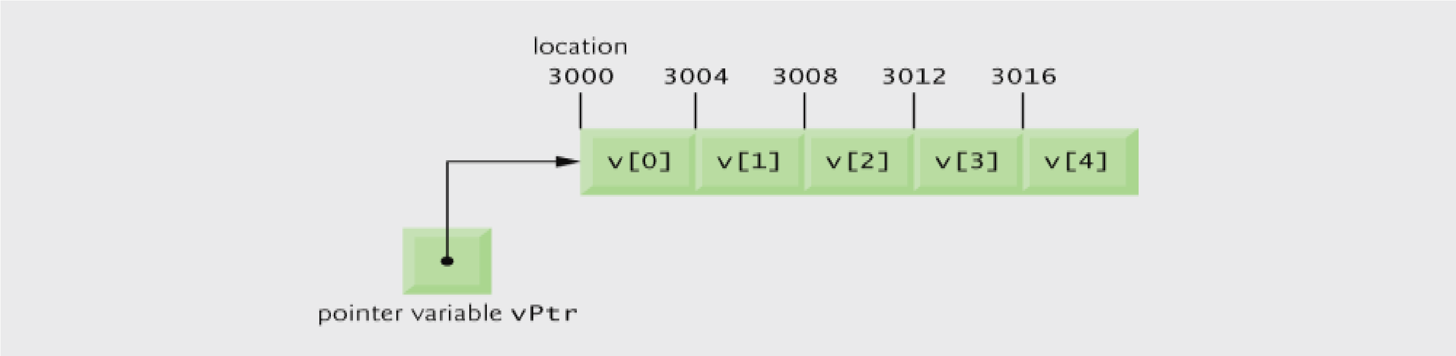
- fig07_15.c (2 of 3)

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

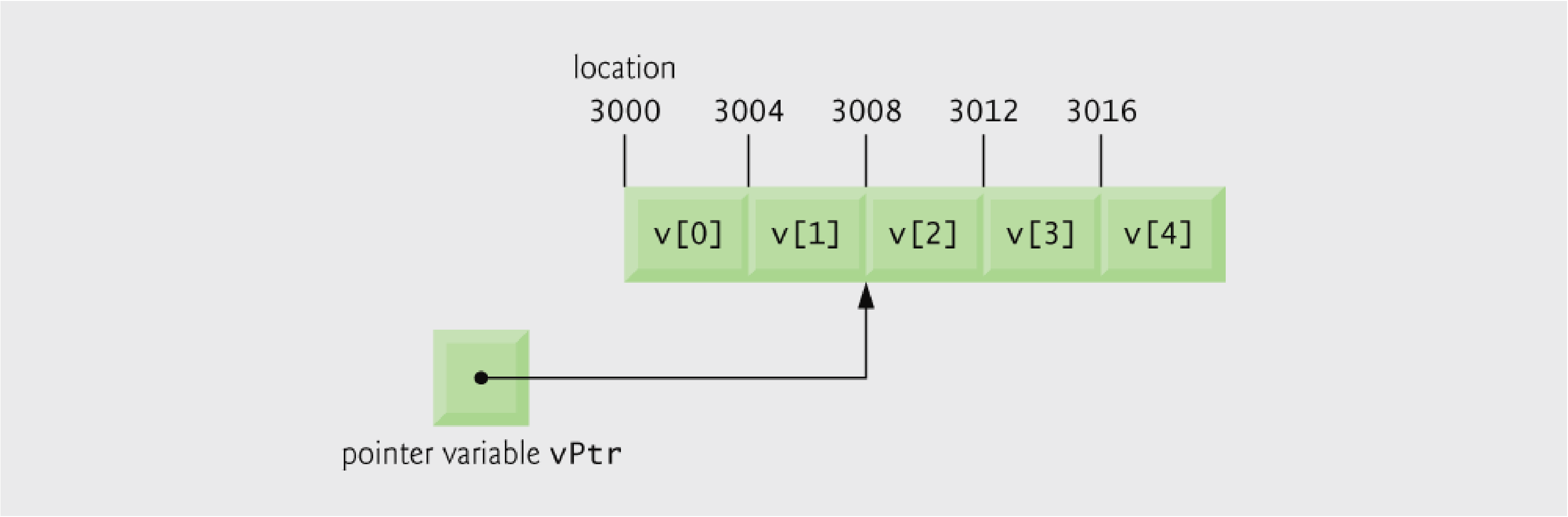
Pointer Expressions and Pointer Arithmetic

- 5 element `int` array on machine with 4 byte `ints`
  - `vPtr` points to first element `v[ 0 ]`
    - at location 3000 `(vPtr = 3000)`
  - `vPtr += 2;` sets `vPtr` to 3008
    - `vPtr` points to `v[ 2 ]` (incremented by 2), but the machine has 4 byte `ints`, so it points to address 3008



| Array **v** and a pointer variable **vPtr** that points to **v**.

The pointer `vPtr` after pointer arithmetic



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

    ```
    vPtr2 = v[ 2 ];
    vPtr = v[ 0 ];
    ```

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to **0**

Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to `void` (type `void *`)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to `void` pointer
    - `void` pointers cannot be dereferenced

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

    - The array name (b) is actually the address of first element of the array `b[ 5 ]`

    ```
    bPtr = &b[ 0 ]
    ```

    - Explicitly assigns `bPtr` to address of first element of b

The Relationship Between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where n is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself

    `*( b + 3 )`

- fig07_20.c (1 of 3)

```
/* Fig. 7.20: fig07_20.cpp
   Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main( void )
{
   int b[] = { 10, 20, 30, 40 }; /* initialize array b */
   int *bPtr = b;                /* set bPtr to point to array b */
   int i;                        /* counter */
   int offset;                   /* counter */

   /* output array b using array subscript notation */
   printf( "Array b printed with:\nArray subscript notation\n" );

   /* loop through array b */
   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   } /* end for */

   /* output array b using array name and pointer/offset notation */
   printf( "\nPointer/offset notation where\n"
           "the pointer is the array name\n" );

   /* loop through array b */
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
   } /* end for */

```

Array subscript notation

Pointer/offset notation

```
    /* output array b using bPtr and array subscript notation */
    printf( "\nPointer subscript notation\n" );

    /* loop through array b */
    for ( i = 0; i < 4; i++ ) {
       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
    } /* end for */

    /* output array b using bPtr and pointer/offset notation */
    printf( "\nPointer/offset notation\n" );

    /* loop through array b */
    for ( offset = 0; offset < 4; offset++ ) {
       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
    } /* end for */

    return 0; /* indicates successful termination */

 } /* end main */
```

Pointer subscript notation

Pointer offset notation

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40
```

(continued on next slide…)

- fig07_20.c
(2 of 3)

(continued from previous slide…)

```
Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

- fig07_20.c (3 of 3)

```c
/* Fig. 7.21: fig07_21.c
   Copying a string using array notation and pointer notation. */
#include <stdio.h>

void copy1( char * const s1, const char * const s2 ); /* prototype */
void copy2( char *s1, const char *s2 ); /* prototype */

int main( void )
{
   char string1[ 10 ];          /* create array string1 */
   char *string2 = "Hello";     /* create a pointer to a string */
   char string3[ 10 ];          /* create array string3 */
   char string4[] = "Good Bye"; /* create a pointer to a string */

   copy1( string1, string2 );
   printf( "string1 = %s\n", string1 );

   copy2( string3, string4 );
   printf( "string3 = %s\n", string3 );

   return 0; /* indicates successful termination */

} /* end main */

```

- fig07_21.c (1 of 2)

```c
/* copy s2 to s1 using array notation */
void copy1( char * const s1, const char * const s2 )
{
   int i; /* counter */

   /* loop through strings */
   for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
      ; /* do nothing in body */
   } /* end for */

} /* end function copy1 */

/* copy s2 to s1 using pointer notation */
void copy2( char *s1, const char *s2 )
{
   /* loop through strings */
   for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
      ; /* do nothing in body */
   } /* end for */

} /* end function copy2 */
```

Condition of **`for`** loop actually performs an action

```
string1 = Hello
string3 = Good Bye
```

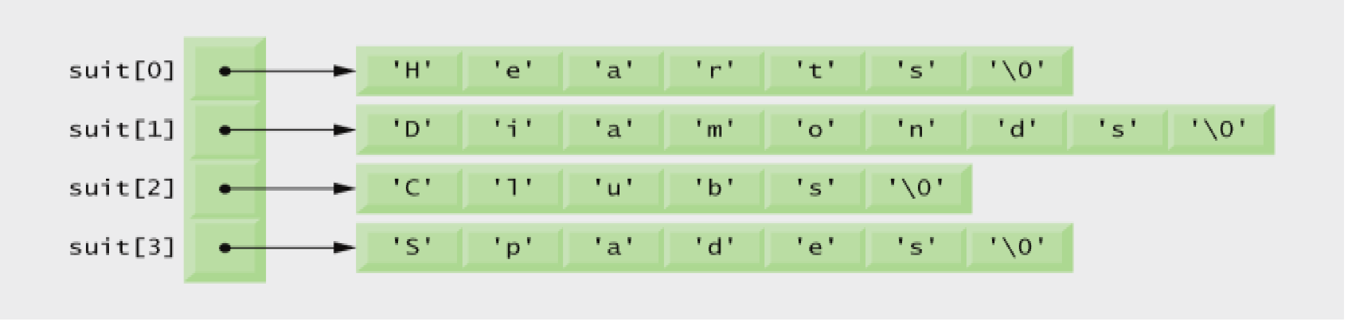
- fig07_21.c (2 of 2)

Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
  "Clubs", "Spades" };
```

  – Strings are pointers to the first character
  – `char *` – each element of `suit` is a pointer to a `char`
  – The strings are not actually stored in the array `suit`, only pointers to the strings are stored



Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to Functions

- Example: bubblesort
  - Function `bubble` takes a function pointer
    - `bubble` calls this helper function
    - this determines ascending or descending sorting
  - The argument in `bubble` for the function pointer:

    ```
    int ( *compare )( int a, int b )
    ```

    tells `bubble` to expect a pointer to a function that takes two `int`s and returns an `int`
  - If the parentheses were left out:

    ```
    int *compare( int a, int b )
    ```

    - Defines a function that receives two integers and returns a pointer to a `int`

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order;   /* 1 for ascending order or 2 for descending order */
   int counter; /* counter */

   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );

   printf( "\nData items in original order\n" );

   /* output original array */
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */

```

bubble function takes a function pointer as an argument

- fig07_26.c (1 of 4)

```c
   /* sort array in ascending order; pass function ascending as an
     argument to specify ascending sorting order */
   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nData items in ascending order\n" );
   } /* end if */
   else { /* pass function descending */
      bubble( a, SIZE, descending );
      printf( "\nData items in descending order\n" );
   } /* end else */

   /* output sorted array */
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */

```

depending on the user's choice, the **`bubble`** function uses either the **`ascending`** or **`descending`** function to sort the array

- fig07_26.c (2 of 4)

```c
/* multipurpose bubble sort; parameter compare is a pointer to
   the comparison function that determines sorting order */
void bubble( int work[], const int size, int (*compare)( int a, int b ) )
{
   int pass;  /* pass counter */
   int count; /* comparison counter */

   void swap( int *element1Ptr, int *element2ptr ); /* prototype */

   /* loop to control passes */
   for ( pass = 1; pass < size; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( count = 0; count < size - 1; count++ ) {

         /* if adjacent elements are out of order, swap them */
         if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
            swap( &work[ count ], &work[ count + 1 ] );
         } /* end if */

      } /* end for */

   } /* end for */

} /* end function bubble */

```

- fig07_26.c (3 of 4)

Note that what the program considers "out of order" is dependent on the function pointer that was passed to the **bubble** function

```c
/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold; /* temporary holding variable */

   hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */

/* determine whether elements are out of order for an ascending
   order sort */
int ascending( int a, int b )
{
   return b < a; /* swap if b is less than a */

} /* end function ascending */

/* determine whether elements are out of order for a descending
   order sort */
int descending( int a, int b )
{
   return b > a; /* swap if b is greater than a */

} /* end function descending */
```

- fig07_26.c (4 of 4)

Passing the **bubble** function **ascending** will point the program here

Passing the **bubble** function **descending** will point the program here

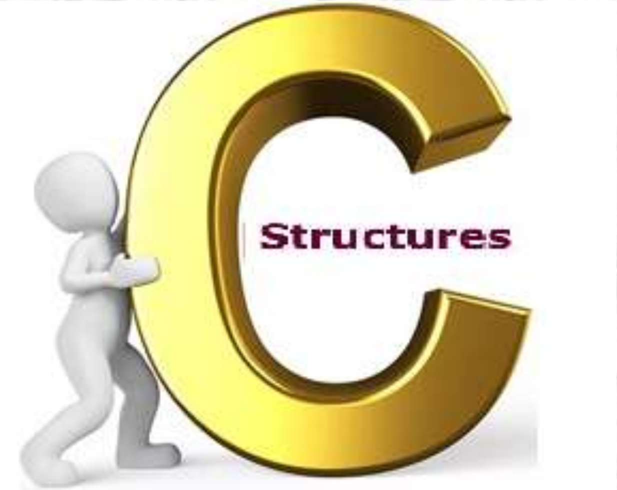
```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Module-5
Structures in C



Prof. Swetha M S
Assistant Professor, Dept. of ISE
BMS Institute of Technology & Mgmt.
Bengaluru.

Structures

Defn:

- Is a user defined data type used to store values of different data types under a common name .
- Collection of data of same or different data type

Eg:

- Library info: (Acc no, title, author, pub, price..)
- Student info: (rno, name, DOB, addr, marks..)

Declaring Structures

```
struct tag
 {
  //member declarations;
   data_type member_var1;
   data_type member _var2;
   ….
 };
```

Eg:

```
struct person
{
 char name[10];
 int  age;
};
```

Defining structure variable

Note:

- Declaration of structure only list its members, it does not allocate any memory for member variables.
- Memory is allocated only when member variables were defined.

Syntax 1:

struct tag var_list;

Eg:

struct person p1,p2;

Syntax 2:

```
   struct tag
{
  member declarations;
} var_list;
```

Eg:

```
struct person
{
  char name[10];
  int age;
} p1,p2;
```

Note:

It is possible to omit structure name while declaration. Such structures are called anonymous structures.

Eg:

```
struct
{
  char name[10];
  int age;
} p1,p2;
```

Initializing structure variables

- Each structure variable contains a copy of all members of the structure.

1. struct person p1={“ bala”, 24};

Accessing structure members

- We can access structure members using dot operator(.)

Syntax

struct_var . member name

2. Second approach of initializing structure variables

struct person p1;

strcpy(p1.name,"bala");

P1.age=24;

Eg pgm

```
#include<stdio.h>
main()
{
  struct student
  {
    int rno;
    char name[10];
     int age;
    char grade;
  } s1;
 s1.rno=101;
strcpy(s1.name,"james");
s1.age=24;
s1.grade='A';
printf("\n student details\n");
printf("Rollno: %d",s1.rno);
printf("Name: %s", s1.name);
printf("Age: %d", s1.age);
printf("Grade: %c ", s1.grade);
return 0;
}
```

OUTPUT:

student details

Rollno: 101

Name: james

Age: 24

Grade:A

Eg pgm

```
#include<stdio.h>
main()
{
  struct student
  {
    int rno;
    char name[10];
     int age;
    char grade;
  } s1;
printf("Enter Roll No:\n");
scanf("%d", &s1.rno);
printf("Enter Name :\n");
gets( s1.name);
printf("Enter Age:\n");
scanf("%d", &s1.age);
printf("Enter Grade:\n");
scanf("%c", s1.grade);
printf("\n student details\n");
printf("Rollno: %d",s1.rno);
printf("Name: %s", s1.name);
printf("Age: %d", s1.age);
printf("Grade: %c ", s1.grade);
return 0;
}
```

OUTPUT:

student details

Rollno: 101

Name: james

Age: 24

Grade: A

Nested Structures

- A structure inside another structure is called nested structure.

2 ways

1. The complete definition of a structure is placed inside the definition of another structure.
2. Structures are defined separately and variable of structure type is declared inside another structure.

Nested Structures

Ist Approach

```
struct student
{
    int rno;
    char name[20];
    struct date
    {
      int day, month, year;
     } dob;
} stud1;
```

2nd Approach

```
struct date
{
int day,month,year;
};
struct student
{
     int rno;
    char name[20];
    struct date dob;
} stud1;
```

Accessing nested structure

- If a structure A has another Structure B nested inside it

```
struct A
{
    struct B
    {
     }b;
 }a;
```

Then b data members can be accessed by **a.b.data member**

Eg

```
#include<stdio.h>
struct Address
{ char HouseNo[25];
char City[25];
char PinCode[25];
};
struct Employee
{ int Id;
char Name[25];
float Salary;
struct Address Add;
};
```

```
void main()
{
 int i;
 struct Employee E;
 printf("\n\tEnter Employee Id : ");
 scanf("%d", &E.Id);
 printf("\n\tEnter Employee Name : ");
 scanf("%s", &E.Name);
 printf("\n\tEnter Employee Salary : ");
scanf("%f", &E.Salary);
```

```
printf("\n\tEnter Employee House No : ");
scanf("%s",&E.Add.HouseNo);
printf("\n\tEnter Employee City : ");
scanf("%s",&E.Add.City);
printf("\n\tEnter Employee House No : ");
scanf("%s",&E.Add.PinCode);
printf("\nDetails of Employees");
printf("\n\tEmployee Id : %d",E.Id);
printf("\n\tEmployee Name : %s",E.Name);
printf("\n\tEmployee Salary : %f",E.Salary);
printf("\n\tEmployee House No : %s",E.Add.HouseNo);
printf("\n\tEmployee City : %s",E.Add.City);
printf("\n\tEmployee House No : %s",E.Add.PinCode);
}
```

OUTPUT

Enter Employee Id : 101
Enter Employee Name : Suresh
Enter Employee Salary : 45000
Enter Employee House No : 4598/D
Enter Employee City : Delhi
Enter Employee Pin Code : 110056
Details of Employees
Employee Id : 101
Employee Name : Suresh
Employee Salary : 45000
Employee House No : 4598/D
Employee City : Delhi
Employee Pin Code : 110056

Arrays of Structure

- Arrays of structure type is required when you need to apply the same structure to a set of objects.

Syntax

```
struct student
{
  int rno;
  char name[20];
} stud[3];  // arrays of structure
```

```
#include <stdio.h>
struct student
{
   char name[50];
    int roll;
    float marks;
};

int main()
{
struct student s[10];      // Array of structure
int i;
printf("Enter information of students:\n");
for(i=0;i<10;++i)
    {
     s[i].roll=i+1;
     printf("\nFor roll number %d\n",s[i].roll);
```

```
printf("Enter name: ");
scanf("%s",s[i].name);
printf("Enter marks: ");
scanf("%f",&s[i].marks);
printf("\n");
}
printf("Displaying information of students:\n\n");
for(i=0;i<10;++i)
{
   printf("\nInformation for roll number %d:\n",i+1);
    printf("Name: ");
    puts(s[i].name);
 printf("Marks: %.1f",s[i].marks);
}
return 0;
 }
```

OUTPUT for Arrays of structure

Enter information of students:
For roll number 1
Enter name: Tom
Enter marks: 98

For roll number 2
Enter name: Jerry
Enter marks: 89
.
.
Displaying information of students:
Information for roll number 1:
Name: Tom
Marks: 98 . . .

Structures and functions

In C, structure can be passed to functions by two methods:

- Pass by value (passing actual value as argument)
- Pass by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as normal variable.

If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

Passing structure by reference

The address location of structure variable is passed to function while passing it by reference.

If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Passing structure by value

```
#include <stdio.h>
struct student
{ char name[50]; int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration otherwise
compiler shows error */
int main()
{  struct student s1;
    printf("Enter student's name: ");
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
     Display(s1);     // passing structure variable s1 as argument
     return 0;
}
```

```
void Display(struct student stu)
{
printf("\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);
}
```

OUTPUT

```
Enter student's name: Kevin
Enter roll number: 149
Name: Kevin
Roll: 149
```

Passing structure by reference

```
#include <stdio.h>
struct distance
{ int feet;
float inch; };
 void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{ struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d",&dist1.feet);
```

```
printf("Enter inch: ");
scanf("%f",&dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d",&dist2.feet);
printf("Enter inch: ");
scanf("%f",&dist2.inch);
Add(dist1, dist2, &dist3);
/*passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by reference */
printf("\nSum of distances = %d\'-%.1f \"",dist3.feet, dist3.inch);
return 0;
}
```

```
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
    /* Adding distances d1 and d2 and storing it in d3 */
    d3->feet=d1.feet+d2.feet;
    d3->inch=d1.inch+d2.inch;
    if (d3->inch>=12)
      { /* if inch is greater or equal to 12, converting it to feet. */
          d3->inch-=12;
          ++d3->feet;
      }
}
```

OUTPUT

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18'-2.3"

typedef

- Reserved keyword in c
- It allows you to create a new data type name for an existing data type.
- We can create a new data type name for primitive as well as user defined data type.

Syntax:

```
typedef old_data_type new_data_type;
```

typedef

Eg1:

```
typedef int integer;
 integer a,b,c;
 integer a[10];

Typedef int raju
raju a;
Raju b;
```

Eg2:

```
struct emp
{
 int emp_id;
 char name[10];
 };
typedef struct emp employee;
employee emp1,emp2;
```

Typedef eg

```
#include <stdio.h>
#include <string.h>
 typedef struct Books
{
char title[50];
char author[50];
char subject[100];
 int book_id;
} Book[10];
```

```
int main( )
{
Book book[10];
strcpy( book.title, "C Programming");
strcpy( book.author, "Nuha Ali");
strcpy( book.subject, "C Programming”);
book.book_id = 6495407;
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n",book.book_id);
 return 0;
}
```

typedef vs #define

- **#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences –
  - **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, you can define 1 as ONE etc.
  - **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

Unions

- Unions are similar to structures used to store values of different data types.

```
union tag_name
{
 data type1 var1;
 data_type2 var2;
.
.
};
```

Structure	Union
1.The keyword **struct** is used to define a structure	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.

eg

```
struct stu
{
        char c;
        int l;
        float p;            // Total size: 7 bytes
};
union emp
{
        char c[20];
        int l;
        float p;      // total bytes : 4 bytes
};
```

Lab Program -13. STRUCTURES

Implement structures to read, write, compute average-marks and the students scoring above and below the average marks for a class of N students

1) Program to maintain a record of student using structure

```
#include <stdio.h>
struct student
{
char usn[50];
char name[50];
int marks;
} s[10];
```

```
void main()
{
int i,n,countav=0,countbv=0; float sum,average;
clrscr();
printf("Enter number of Students\n"); scanf("%d",&n);
printf("Enter information of students:\n");
```

2)Storing information

```
for(i=0;i<n;i++)
{
printf("Enter USN: ");
scanf("%s",s[i].usn);
printf("Enter name: ");
scanf("%s",s[i].name);
printf("Enter marks: ");
scanf("%d",&s[i].marks);
printf("\n");
}
```

3)displaying information

```
printf("Displaying Information:\n\n");
for(i=0; i<n; i++)
{
printf("\nUSN: %s\n",s[i].usn); printf("Name: ");
puts(s[i].name);
printf("Marks: %d",s[i].marks); printf("\n");
}
for(i=0;i<n;i++)
{
sum=sum+s[i].marks;
}
average=sum/n;
printf("\nAverage marks: %f",average);
```

```
countav=0;
countbv=0;
for(i=0;i<n;i++)
{
if(s[i].marks>=average)
countav++;
else
}
countbv++;
printf("\nTotal No of students above average= %d",countav);
printf("\nTotal No of students below average= %d",countbv);
}
```

Output

<table>
<tr>
<td>1</td>
<td>N=3, Enter the student details
<table>
<tr><td></td><td>USN</td><td>Name</td><td>Marks</td></tr>
<tr><td>S[0]</td><td>1RN18CS001</td><td>Chetan</td><td>500</td></tr>
<tr><td>S[1]</td><td>1RN18CS002</td><td>Darshan</td><td>510</td></tr>
<tr><td>S[2]</td><td>1RN18CS003</td><td>Pallavi</td><td>550</td></tr>
</table>
</td>
<td>Average marks:520

Total No of students above average=1

Total No of students below average=2</td>
<td>Average marks:520

Total No of students above average=1

Total No of students below average=2</td>
</tr>
</table>

Module-5

Prof. Swetha M S
Assistant Professor, Dept. of ISE
BMS Institute of Technology & Mgmt.
Bengaluru.

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Character pointer and functions
- Pointer to pointer

POINTERS

1 POINTER

Pointer definition: Pointer is a variable that holds the address of another variable.

Declaration and Initilization of pointers

The operators used to represent pointers are:

- Address operator (&)
- Indirection operator (*)

Syntax:

```
ptr_data_type   *ptr_variable_name
ptr_variable_name  =  &variable_name
```

where variable_name is the variable whose address has to be stored in pointer.

Example:

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

that is ptr is a pointer holding address of variable 'a' and *ptr holds the value of a.

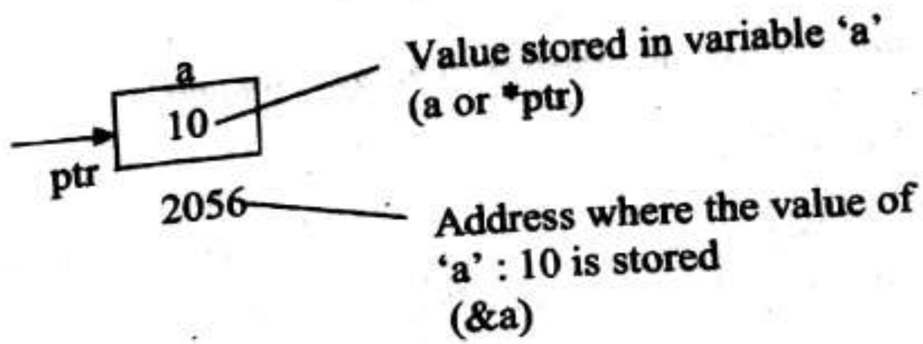
that is ptr is a pointer

Example program:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a=10;
        int *ptr;
        ptr=&a;
        printf("%d\n",  a);
        printf("%d\n", &a);
        printf("%d\n",  ptr);
        printf("%d\n", *ptr);
        getch( );
}
```

Output:

10
2056
2056
10



```c

   /*Using the & and * operators */
#include <stdio.h>

int main( void )
{
   int a;         /* a is an integer */
   int *aPtr;     /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;     /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

- fig07_04.c (1 of 2)

If **aPtr** points to **a**, then **&a** and **aPtr** have the same value.

a and ***aPtr** have the same value

&*aPtr and ***&aPtr** have the same value

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

- fig07_04.c (2 of 2)

Pointers and functions

2.1. Pointers and functions (call by reference)

Call by reference method involves use of address of variables as actual parameters in calling function and pointer variables with (*) indirection operator is used at called function to perform required operations that is as formal parameters.

Consider an example of swapping two numbers using call by reference or using pointers

```
#include <stdio.h>
#include<conio.h>
void swap (int *a, int *b);
void main( )
{
            int x =10, y = 20;
            swap(&x, &y);
            printf("after swapping:\nx=%d\ny=%d", x,y);
            getch( );
}
```

```
void swap (int *a, int *b)
{
        int  temp;
        temp = *a;
        *a = *b;
        *b = temp;
}
```

Output:

```
after swapping:
x=20
y=10
```

Here instead of passing actual values of x and y pointers, address of x and y are passed.

Cube a variable using call-by-value

```
#include <stdio.h>
int cubeByValue( int n );   /* prototype */
int main( void )
{
      int number = 5;
      printf( "The original value of number is %d", number );
   /* pass number by value to cubeByValue */
       number = cubeByValue( number );
        printf( "\nThe new value of number is %d\n", number );
        return 0;
}
    int cubeByValue( int n )
   {
        return n * n * n;
  }
```

The original value of number is 5
The new value of number is 125

Call-by-reference with a pointer argument

Function prototype takes a pointer argument

```
#include <stdio.h>
void cubeByReference( int *nPtr ); /* prototype */
int main( void )
{
  int number = 5;
printf( "The original value of number is %d", number );
/* pass address of number to cubeByReference */
    cubeByReference( &number );
printf( "\nThe new value of number is %d\n", number );
 return 0;
    }
void cubeByReference( int *nPtr )
{
 *nPtr = *nPtr x*nPtr x *nPtr;  /* cube *nPtr */
}
```

Function **`cubeByReference`** is passed an address, which can be the value of a pointer variable

In this program, **`*nPtr`** is **`number`**, so this statement modifies the value of **`number`** itself.

Pointers and array

2.2. Pointers and arrays

The operations performed using array concept can also be done using pointers.

Syntax:

```
data_type *ptr_name;
ptr_name = &array_name or ptr_name = array_name;
```

Here pointer does not point to all the elements of an array, instead initially it points to the first element of an array later which is incremented to get other elements.

Example:
```
int a[10] = {11,12,13,14};
int * ptr;
ptr= &a or ptr=a;
```
here ptr is pointing to 11 initially.

It can be explained using below program

```
/* program to demonstrate pointers to arrays concept */
#include <stdio.h>
#include<conio.h>
void main( )
{
        int a[10]={11,12,13,14};
        int *ptr;

ptr=a;   /*initially pointing to first element 11*/
for (i=0, i<4; i++)          /* four elements*/
{
    printf("%d\t", a[i]);
    printf("%d\n", &a[i]);
    printf("%d\t", *ptr);
    printf("%d\t", ptr);
    ptr++;
    /* making ptr to point next value by doing ptr =
    ptr+1*/
}
getch( );
```

}

Output:

```
11   2056   /*value a[0] pointed by ptr at first iteration and address where that value is stored*/
11   2056   /*value *ptr pointed by ptr at first iteration and address where that value is stored*/

12   2058   /*value a[1] pointed by ptr at second iteration and address where that value is stored*/
12   2058   /*value *ptr pointed by ptr at first iteration and address where that value is stored*/

13   2060   /*value a[2] pointed by ptr at third iteration and address where that value is stored*/
13   2060   /*value *ptr pointed by ptr at first iteration and address where that value is stored*/

14   2062   /*value a[3] pointed by ptr at fourth iteration and address where that value is stored*/
14   2062   /*value *ptr pointed by ptr at first iteration and address where that value is stored*/
```

position	0	1	2	3
values	11	12	13	14
adddress	2056	2058	2060	2062

Initial position of ptr

since integer type occupies 2 bytes for each element. If it reserves 2056 for first element as starting address, 2056+2=2058 as second element's starting address, 2058+2=2060 for third element and so on.

Pointers to Strings

2.3. Character pointer and functions or Pointers to the strings

Strings are array of characters instead of integer values of array, here pointer points to the character present in string represented as an array.

Syntax:
```
data_type  &ptr_name;
ptr_name = string_name;
```

Example:
```
char str[20] = "america"
char *ptr;
ptr = str;
```

Here pointer does not point to all the character of a string, instead initially it points to the first element or first character of a string later which is incremented to get other elements. It can be explained using below program.

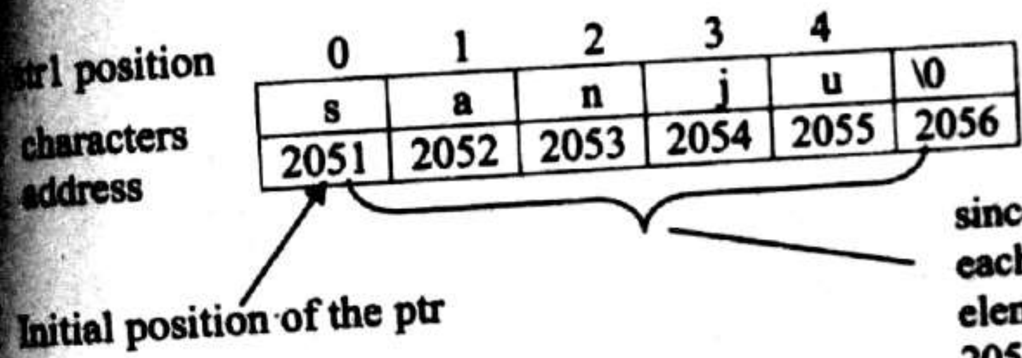
/*string copy using pointer to string concept (using single pointer) */
```
#include<stdio.h>
#include<conio.h>
```

```
id main( )

  int i;
  char str1[20]= "sanju";
  char str2[20];
  char *ptr;
  ptr = str1;
  for (i=0; str1[i]!='\0'; i++)
 {
       str2[i]=*ptr;
       ptr++;
 }
 str2[i]='\0';
 printf("string2 after copying is %s", str2);
 getch( );
}
```

Output:

tring2 after copying is nanju



since character type occupies 1 bytes for each element. If it reserves 2051 for first element as starting address, 2051+1=2052 as second element's starting address, 2052+1=2053 for third element and so on.

Pointer to Pointer

2.4. Pointer to pointer

Pointer is a variable that stores the address of another variable. Pointer storing the address of another pointer, that is pointer pointing to another pointer is called as pointer to pointer.

Declaration:

data_type **pointer_name;

Initialization:

pointer_name = &another_pointer_name

The below example demonstrates pointer to pointer concept:

```
int a = 10;
int *ptr1, **ptr2;
ptr1 = &a;
ptr2 = &ptr1;            /*ptr2 is the pointer to the another pointer ptr1*/
```

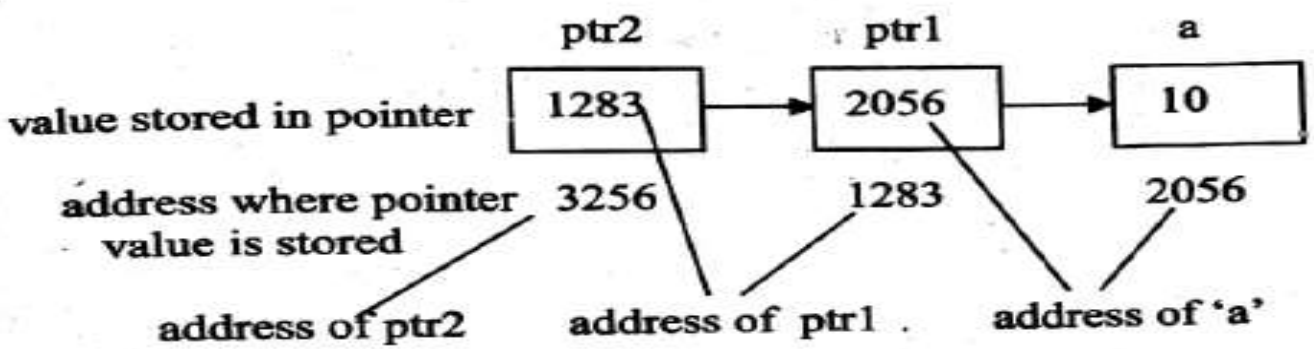


Figure 1: Pointer to pointer

```c
#include <stdio.h>
#include<conio.h>
void main( )
{
      int  a=10;
      int *ptr1, **ptr2;
      ptr1 = &a;
      ptr2 = &ptr1;
      printf("%d\n", a);
      printf("%d\n", &a);
      printf("%d\n", ptr1);
      printf("%d\n", &ptr1);
      printf("%d\n", *ptr1);
      printf("%d\n", ptr2);
      printf("%d\n", *ptr2);
      printf("%d\n", **ptr2);
      getch( );
}
```

output

```
10
2056
2056
1283
10
1283
2056
10
```

Address arithmetic

2.5. Address Arithmetic

1. An integer value can be added or subtracted from a pointer. It can be incremented or decremented.

Array

11	9	8	14	a
2050	2052	2054	2056	Address

2 bytes difference since it is integer

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

```
void main( )
{
    int  a[4]={11,9,8,14};


   int *ptr
   ptr = &a;                                                 output
   printf("%d\n", ptr);                        ——→ 2050
   printf("%d\n", *ptr );                      ——→ 11
   ptr++;  /* we can use ptr=ptr+1 or ptr+=1 */
   printf("%d\n", ptr );                       ——→ 2052
   printf("%d\n",* ptr );                      ——→ 9
   ptr--; /* we can use ptr=ptr-1 or ptr- =1*/
   printf("%d\n", ptr);                        ——→ 2050
   printf("%d\n", *ptr );                      ——→ 11
   getch( );
}
```

2. If two pointers pointing to same type of data then one pointer value can be assigned to another.

Example:
```
int *ptr1, *ptr2;
ptr1=ptr2;
```

3. Pointer can be assigned a Null

Example:
```
int *ptr1;
ptr1=NULL;
```

4. Subtraction of two pointer variables can be performed when both are pointing to elements of same array.

5. Two pointers cannot be multiplied added or dIvided directly

6. Relational operators can be used between the pointer

Example:
```
int * ptr1,  * ptr2;
ptr1>ptr2 , ptr1=ptr2, ptr1<ptr2 etc.
```

Advantages and disadvantages

2.6. Advantages and disadvantages of pointers

Advantages of pointers in C

- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allows us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allows us to resize the dynamically allocated memory block.

Disadvantages of pointer in C

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

2.7. Programming examples

Example 1: Write a C program to read n elements to an array and print those elements using pointer to an array.

```
#include <stdio.h>
#include<conio.h>
void main( )
{
        int a[100], *ptr, i;
        printf("enter number of elements.\n");
        scanf("%d", &n);
        ptr=a;
        for (i=0; i<n; i++)
        {
                scanf("%d", ptr);
                /* no & symbol since ptr itself an address*/
                ptr++;

        }
```

```
    }
    ptr=a;
    /* initialize ptr back to first element*/
    printf("array elements are");
    for (i=0; i<n; i++)
    {
                printf("%d\t", *ptr);
                ptr++;

    }
    getch( );
}
```

Output:

```
enter number of elements
5
11    12    13    14    15
Array elements are
11    12    13    14    15
```

Lan Program 14- USE OF POINTERS

Develop a program using pointers to compute the sum, mean and standard deviation of all elements stored in an array of n real numbers

The formula for standard deviation (SD)

$$\text{Mean }(\bar{x}) = \frac{\sum x}{n}$$

$$\text{SD} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

Standard Deviation

$$\sigma = \sqrt{\frac{(\bar{x} - x_1)^2 + (\bar{x} - x_2)^2 + \ldots + (\bar{x} - x_n)^2}{n}}$$

- **Step 1:** Find the mean.
- **Step 2:** For each data point, find the square of its distance to the mean.
- **Step 3:** Sum the values from Step 2.
- **Step 4:** Divide by the number of data points.
- **Step 5:** Take the square root.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
float a[10], *ptr, mean, std, sum=0, sumstd=0;
int n,i;
clrscr();
printf("Enter the no of elements\n"); scanf("%d",&n);
printf("Enter the array elements\n"); for(i=0;i<n;i++)
{
scanf("%f",&a[i]);
}
ptr=a;
```

```
for(i=0;i<n;i++)
{
sum=sum+ *ptr; ptr++;
}
mean=sum/n; ptr=a;
for(i=0;i<n;i++)
{
sumstd=sumstd + pow((*ptr - mean),2);
ptr++;
}
std= sqrt(sumstd/n);
printf("Sum=%.3f\t",sum); printf("Mean=%.3f\t",mean);
printf("Standard deviation=%.3f\t",std);
return 0;
}
```

14.2 Test Cases

Test No	Input Parameters	Expected output	Obtained output	Remarks
1	N=5 Array elements 1 5 9 6 7	Sum=28 Mean=5.6 Standard deviation=2.09	Sum=28.000 Mean=5.600 Standard deviation=2.098	PASS
2	N=4 Array elements **2.3 1.1 4.5 2.78**	Sum=10.68 Mean=2.67 Standard deviation=0.863	Sum=10.680 Mean=2.670 Standard deviation=0.863	PASS
Test for the following cases and Records				
3	N=5 Array elements 2 3 4 8 10			
4	N=6 Array elements 2.4 5.6 2.0 4.12 5.17 0.14			

Viva Question

1. Define pointer?
2. How do you declare a pointer variable?
3. What is * and &in pointer concept.
4. What are the advantages and disadvantages of using pointer?
5. Give the difference between static allocation and dynamic allocation of memory space.
6. What is the effect of the ++ and --operators on pointer variable
7. Explain the pointers to arrays concept?

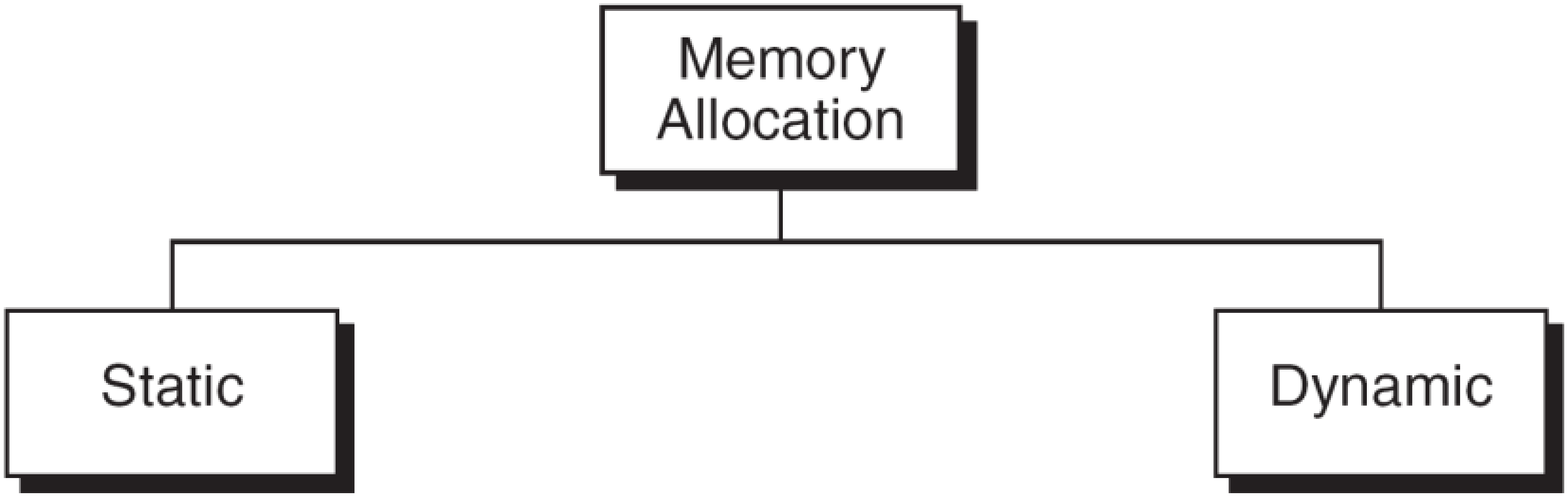
Module-5
Dynamic memory allocation

Prof Swetha M S

Assistant Professor

ISE-BMSIT&M

Types of memory allocation



Static vs Dynamic Memory

Definition	
Static memory allocation is a method of allocating memory, and once the memory is allocated, it is fixed.	Dynamic memory allocation is a method of allocating memory, and once the memory is allocated, it can be changed.
Modification	
In static memory allocation, it is not possible to resize after initial allocation.	In dynamic memory allocation, the memory can be minimized or maximize accordingly.
Implementation	
Static memory allocation is easy to implement.	Dynamic memory allocation is complex to implement.
Speed	
In static memory, allocation execution is faster than dynamic memory allocation.	In dynamic memory, allocation execution is slower than static memory allocation.
Memory Utilization	
In static memory allocation, cannot reuse the unused memory.	Dynamic memory allocation allows reusing the memory. The programmer can allocate more memory when required . He can release the memory when necessary.

Disadvantages of static memory allocation

- The exact size of array is unknown untill the compile time.
- The size of array you have declared initially can be sometimes insufficient and sometimes more than required.

Dynamic memory allocation

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Functions defined in (stdlib.h)

- malloc-stands for memory allocation.
- calloc-stands for contiguous allocation.
- realloc-stands for reallocation
- free- to release the space

Dynamic memory allocation Functions

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc ()

- The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

Syntax

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

Eg:

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

calloc ()

- The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax

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

Eg:

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

realloc ()

- If the previously allocated memory using malloc and calloc is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc()

Syntax

- ptr=realloc(ptr,newsize);

Eg:

- ptr=realloc(ptr,100*sizeof(char));

free

- Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

Syntax

- free(ptr);

Example pgm using malloc

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
 int main()
 { char *mem_alloc;     /* memory allocated dynamically */
  mem_alloc = malloc( 15 * sizeof(char) );
  if(mem_alloc== NULL )
   {
     printf("Couldn't able to allocate requested memory\n");
    }
   else
  {
  strcpy( mem_alloc,“hai hello ");
    }
printf("Dynamically allocated memory content : %s\n", mem_alloc );
free(mem_alloc);
}
```

O/P

- Dynamically allocated memory content: hai hello

Example pgm using calloc

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
 int main()
 { char *mem_alloc;     /* memory allocated dynamically */
  mem_alloc = calloc( 15, sizeof(char) );
  if(mem_alloc== NULL )
   {
     printf("Couldn't able to allocate requested memory\n");
    }
   else
  {
  strcpy( mem_alloc,“hai hello every one");
    }
printf("Dynamically allocated memory content : %s\n", mem_alloc );
free(mem_alloc);
}
```

O/P

- Dynamically allocated memory content: hai hello every one

Example program for realloc

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
 int main()
{ char *mem_alloc;       /* memory allocated dynamically */
mem_alloc = malloc( 20 * sizeof(char) );
if( mem_alloc == NULL )
{
printf("Couldn't able to allocate requested memory\n");
}
else
 {
strcpy( mem_alloc,“hai hello every one");
}
```

```
printf("Dynamically allocated memory content : " \ "%s\n",
mem_alloc );
mem_alloc=realloc(mem_alloc,100*sizeof(char));
if( mem_alloc == NULL )
 {
printf("Couldn't able to allocate requested memory\n");
 }
else
 {
  strcpy( mem_alloc,"space is extended upto 100 characters");
}
printf("Resized memory : %s\n", mem_alloc );
free(mem_alloc);
 }
```

Output

- Dynamically allocated memory content: hai hello every one
- Resized memory: space is extended upto 100 characters

To find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

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

```
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
 sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```