



JAWAHARLAL COLLEGE OF ENGINEERING AND TECHNOLOGY
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological
University, Kerala)
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(NBA Accredited)



COURSE MATERIAL

EST 102 PROGRAMMING IN C

VISION OF THE INSTITUTION

Emerge as a centre of excellence for professional education to produce high quality engineers and entrepreneurs for the development of the region and the Nation

MISSION OF THE INSTITUTION

- To become an ultimate destination for acquiring latest and advanced knowledge in the multidisciplinary domains.
- To provide high quality education in engineering and technology through innovative teaching-learning practices, research and consultancy, embedded with professional ethics.
- To promote intellectual curiosity and thirst for acquiring knowledge through outcome based education.
- To have partnership with industry and reputed institutions to enhance the employability skills of the students and pedagogical pursuits.
- To leverage technologies to solve the real life societal problems through community services.

ABOUT THE DEPARTMENT

- Established in: 2008
- Courses offered: B.Tech in Computer Science and Engineering
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To produce competent professionals with research and innovative skills, by providing them with the most conducive environment for quality academic and research oriented undergraduate education along with moral values committed to build a vibrant nation.

DEPARTMENT MISSION

- Provide a learning environment to develop creativity and problem solving skills in a professional manner.
- Expose to latest technologies and tools used in the field of computer science.
- Provide a platform to explore the industries to understand the work culture and expectation of an organization.
- Enhance Industry Institute Interaction program to develop the entrepreneurship skills.
- Develop research interest among students which will impart a better life for the society and the nation.

PROGRAMME EDUCATIONAL OBJECTIVES

Graduates will be able to

- Provide high-quality knowledge in computer science and engineering required for a computer professional to identify and solve problems in various application domains.
- Persist with the ability in innovative ideas in computer support systems and transmit the knowledge and skills for research and advanced learning.
- Manifest the motivational capabilities, and turn on a social and economic commitment to community services.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration

for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Outcomes: After the completion of the course the student will be able to

CO 1	Analyze a computational problem and develop an algorithm/flowchart to find its solution
CO 2	Develop readable* C programs with branching and looping statements, which uses Arithmetic, Logical, Relational or Bitwise operators.
CO 3	Write readable C programs with arrays, structure or union for storing the data to be processed
CO 4	Divide a given computational problem into a number of modules and develop a readable multi-function C program by using recursion if required, to find the solution to the computational problem
CO 5	Write readable C programs which use pointers for array processing and parameter passing
CO 6	Develop readable C programs with files for reading input and storing output

readable* - readability of a program means the following:









































1. Logic used is easy to follow
2. Standards to be followed for indentation and formatting
3. Meaningful names are given to variables
4. Concise comments are provided wherever needed

PROGRAM SPECIFIC OUTCOMES (PSO)

The students will be able to

- Use fundamental knowledge of mathematics to solve problems using suitable analysis methods, data structure and algorithms.
- Interpret the basic concepts and methods of computer systems and technical specifications to provide accurate solutions.
- Apply theoretical and practical proficiency with a wide area of programming knowledge, design new ideas and innovations towards research.

Prerequisite: NIL

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1												
CO2												
CO3												
CO4												
CO5												
CO6												

Assessment Pattern

Bloom's Category	Continuous Assessment Tests		End Semester Examination Marks
	Test 1 (Marks)	Test 2 (Marks)	
Remember	15	10	25
Understand	10	15	25
Apply	20	20	40
Analyse	5	5	10

Evaluate			
Create			

Mark distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	50	100	3 hours

Attendance : 10 marks

Continuous Assessment Test 1 (for theory, for 2 hrs) : 20 marks

Continuous Assessment Test 2 (for lab, internal examination, for 2 hrs) : 20 marks

Internal Examination Pattern: There will be two parts; Part A and Part B. Part A contains 5 questions with 2 questions from each module (2.5 modules x 2 = 5), having 3 marks for each question. Students should answer all questions. Part B also contains 5 questions with 2 questions from each module (2.5 modules x 2 = 5), of which a student should answer any one. The questions should not have sub-divisions and each one carries 7 marks.

End Semester Examination Pattern: There will be two parts; Part A and Part B. Part A contains 10 questions with 2 questions from each module, having 3 marks for each question. Students should answer all questions. Part B contains 2 questions from each module of which a student should answer any one. Each question can have maximum 2 sub-divisions and carry 14 marks.

Sample Course Level Assessment Questions

Course Outcome 1 (CO1): Write an algorithm to check whether largest of 3 natural numbers is prime or not. Also, draw a flowchart for solving the same problem.

Course Outcome 2 (CO2): Write an easy to read C program to process a set of n natural numbers and to find the largest even number and smallest odd number from the given set of numbers. The program should not use division and modulus operators.

Course Outcome 3(CO3): Write an easy to read C program to process the marks obtained by n students of a class and prepare their rank list based on the sum of the marks obtained. There are 3 subjects for which examinations are conducted and the third subject is an elective where a student is allowed to take any one of the two courses offered.

Course Outcome 4 (CO4): Write an easy to read C program to find the value of a mathematical function f which is defined as follows. $f(n) = n! / (\text{sum of factors of } n)$, if n is not prime and $f(n) = n! / (\text{sum of digits of } n)$, if n is prime.

Course Outcome 5 (CO5): Write an easy to read C program to sort a set of n integers and to find the number of unique numbers and the number of repeated numbers in the given set of numbers. Use a function which takes an integer array of n elements, sorts the array using the Bubble Sorting Technique and returns the number of unique numbers and the number of repeated numbers in the given array.

Course Outcome 6 (CO6): Write an easy to read C program to process a text file and to print the Palindrome words into an output file.

Model Question paper

QP CODE:

PAGES:3

Reg No:_____

Name :_____

**APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY FIRST SEMESTER B.TECH DEGREE EXAMINATION,
MONTH & YEAR**

Course Code: EST 102

Course Name: Programming in C (Common to all programs)

Max.Marks:100

Duration: 3 Hours

PART A

Answer all Questions. Each question carries 3 Marks

1. Write short note on processor and memory in a computer.
2. What are the differences between compiled and interpreted languages? Give example for each.
3. Write a C program to read a Natural Number through keyboard and to display the reverse of the given number. For example, if "3214567" is given as input, the output to be shown is "7654123".
4. Is it advisable to use *goto* statements in a C program? Justify your answer.
5. Explain the different ways in which you can *declare & initialize* a single dimensional array.
6. Write a C program to read a sentence through keyboard and to display the count of white spaces in the given sentence.
7. What are the advantages of using functions in a program?
8. With a simple example program, explain *scope* and *life time* of variables in C.
9. Write a function in C which takes the address of a single dimensional array (containing a finite sequence of numbers) and the number of numbers stored in the array as arguments and stores the numbers in the same array in reverse order. Use pointers to access the elements of the array.
10. With an example, explain the different modes of opening a file. (10x3=30)

Part B

Answer any one Question from each module. Each question carries 14 Marks

11. (a) Draw a flow chart to find the position of an element in a given sequence, using linear searching technique. With an example explain how the flowchart finds the position of a given element. (10)

(b) Write a pseudo code representing the flowchart for linear searching. (4)

OR

12. (a) With the help of a flow chart, explain the bubble sort operation. Illustrate with an example. (10)

(b) Write an algorithm representing the flowchart for bubble sort. (4)

13. (a) Write a C program to read an English Alphabet through keyboard and display whether the given Alphabet is in upper case or lower case. (6)

(b) Explain how one can use the builtin function in C, *scanf* to read values of different data types. Also explain using examples how one can use the builtin function in C, *printf* for text formatting. (8)

OR

14. (a) With suitable examples, explain various operators in C. (10)

(b) Explain how characters are stored and processed in C. (4)

15. (a) Write a function in C which takes a 2-Dimensional array storing a matrix of numbers and the order of the matrix (number of rows and columns) as arguments and displays the sum of the elements stored in each row. (6)

(b) Write a C program to check whether a given matrix is a diagonal matrix. (8)

OR

16. (a) Without using any builtin string processing function like *strlen*, *strcat* etc., write a program to concatenate two strings. (8)

(b) Write a C program to perform bubble sort. (6)

17. (a) Write a function namely *myFact* in C to find the factorial of a given number. Also, write another function in C namely *nCr* which accepts two positive integer parameters *n* and *r* and returns the value of the mathematical function $C(n,r) = \frac{n!}{r! \times (n-r)!}$. The function *nCr* is expected to make use of the factorial function *myFact*. (10)

(b) What is recursion? Give an example. (4)

OR

18. (a) With a suitable example, explain the differences between a structure and a union in C. (6)

(b) Declare a structure namely *Student* to store the details (*roll number*, *name*, *mark_for_C*) of a student. Then, write a program in C to find the average mark obtained by the students in a class for the subject *Programming in C* (using the field *mark_for_C*). Use array of structures to store the required data (8)

19. (a) With a suitable example, explain the concept of pass by reference. (6)

(b) With a suitable example, explain how pointers can help in changing the content of a single dimensionally array passed as an argument to a function in C. (8)

OR

20. (a) Differentiate between sequential files and random access files? (4)

(b) Using the prototypes explain the functionality provided by the following functions. (10)

rewind()

i. *fseek()*

ii. *ftell()*

iii. *fread()*

iv. *fwrite()*

(14X5=70)

SYLLABUS

Programming in C (Common to all disciplines)

Module 1

Basics of Computer Hardware and Software

Basics of Computer Architecture: processor, Memory, Input& Output devices

Application Software & System software: Compilers, interpreters, High level and low level languages

Introduction to structured approach to programming, Flow chart Algorithms, Pseudo code (*bubble sort, linear search - algorithms and pseudocode*)

Module 2

Program Basics

Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types , Constants, Console IO Operations, printf and scanf

Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence

Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.(Simple programs covering control flow)

Module 3

Arrays and strings

Arrays Declaration and Initialization, 1-Dimensional Array, 2-Dimensional Array

String processing: In built String handling functions (strlen, strcpy, strcat and strcmp, puts, gets)

Linear search program, bubble sort program, simple programs covering arrays and strings

Module 4

Working with functions

Introduction to modular programming, writing functions, formal parameters, actual parameters Pass by Value, Recursion, Arrays as Function Parameters structure, union, Storage Classes, Scope and life time of variables, *simple programs using functions*

Module 5

Pointers and Files

Basics of Pointer: declaring pointers, accessing data through pointers, NULL pointer, array access using pointers, pass by reference effect

File Operations: open, close, read, write, append

Sequential access and random access to files: In built file handling functions (*rewind()*, *fseek()*, *ftell()*, *feof()*, *fread()*, *fwrite()*), simple programs covering pointers and files.

Text Books

1. Schaum Series, Gottfried B.S., Tata McGraw Hill, Programming with C
2. E. Balagurusamy, McGraw Hill, Programming in ANSI C
3. Asok N Kamthane, Pearson, Programming in C
4. Anita Goel, Pearson, Computer Fundamentals

Reference Books

1. Anita Goel and Ajay Mittal, Pearson, Computer fundamentals and Programming in C
2. Brian W. Kernighan and Dennis M. Ritchie, Pearson, C Programming Language
3. Rajaraman V, PHI, Computer Basics and Programming in C
4. Yashavant P, Kanetkar, BPB Publications, Let us C

Course Contents and Lecture Schedule

Module 1: Basics of Computer Hardware and Software		(7 hours)
1.1	Basics of Computer Architecture: Processor, Memory, Input & Output devices	2 hours
1.2	Application Software & System software: Compilers, interpreters, High level and low level languages	2 hours
1.3	Introduction to structured approach to programming, Flow chart	1 hours
1.4	Algorithms, Pseudo Code (<i>bubble sort, linear search - algorithms and pseudocode</i>)	2 hours
Module 2: Program Basics		(8 hours)
2.1	Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types, Constants, Console IO Operations, printf and scanf	2 hours
2.2	Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, sizeof operator, Assignment operators and Bitwise Operators. Operators Precedence	2 hours

2.3	Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements. <i>(Simple programs covering control flow)</i>	4 hours
Module 3: Arrays and strings:		(6 hours)
3.1	Arrays Declaration and Initialization, 1-Dimensional Array, 2-Dimensional Array	2 hours
3.2	String processing: In built String handling functions(<i>strlen, strcpy, strcat and strcmp, puts, gets</i>)	2 hours
3.3	Linear search program, bubble sort program, <i>simple programs covering arrays and strings</i>	3 hours
Module 4: Working with functions		(7 hours)
4.1	Introduction to modular programming, writing functions, formal parameters, actual parameters	2 hours
4.2	Pass by Value, Recursion, Arrays as Function Parameters	2 hours
4.3	structure, union, Storage Classes, Scope and life time of variables, <i>simple programs using functions</i>	3 hours
Module 5: Pointers and Files		(7 hours)
5.1	Basics of Pointer: declaring pointers, accessing data through pointers, NULL pointer, array access using pointers, pass by reference effect	3 hours
5.2	File Operations: open, close, read, write, append	1 hours
5.3	Sequential access and random access to files: In built file handling functions (<i>rewind(), fseek(), ftell(), feof(), fread(), fwrite()</i>), <i>simple programs covering pointers and files.</i>	2 hours

C PROGRAMMING LAB (Practical part of EST 102, Programming in C)

Assessment Method: The Academic Assessment for the Programming lab should be done internally by the College. The assessment shall be made on 50 marks and the mark is divided as follows: Practical Records/Outputs - 20 marks (internal by the College), Regular Lab Viva - 5 marks (internal by the College), Final Practical Exam – 25 marks (internal by the College).

The mark obtained out of 50 will be converted into equivalent proportion out of 20 for CIE computation.

LIST OF LAB EXPERIMENTS

1. Familiarization of Hardware Components of a Computer
2. Familiarization of Linux environment – How to do Programming in C with Linux
3. Familiarization of console I/O and operators in C
 - i) Display “Hello World”
 - ii) Read two numbers, add them and display their sum
 - iii) Read the radius of a circle, calculate its area and display it
 - iv) Evaluate the arithmetic expression $((a - b / c * d + e) * (f + g))$ and display its solution. Read the values of the variables from the user through console.
4. Read 3 integer values and find the largest among them.
5. Read a Natural Number and check whether the number is prime or not
6. Read a Natural Number and check whether the number is Armstrong or not
7. Read n integers, store them in an array and find their sum and average
8. Read n integers, store them in an array and search for an element in the array using an algorithm for Linear Search
9. Read n integers, store them in an array and sort the elements in the array using Bubble Sort algorithm
10. Read a string (word), store it in an array and check whether it is a palindrome word or not.
11. Read two strings (each one ending with a \$ symbol), store them in arrays and concatenate them without using library functions.
12. Read a string (ending with a \$ symbol), store it in an array and count the number of vowels, consonants and spaces in it.
13. Read two input each representing the distances between two points in the Euclidean space, store these in structure variables and add the two distance values.
14. Using structure, read and print data of n employees (*Name, Employee Id and Salary*)
15. Declare a union containing 5 string variables (*Name, House Name, City Name, State and Pin code*) each with a length of C_SIZE (user defined constant). Then, read and display the address of a person using a variable of the union.
16. Find the factorial of a given Natural Number n using recursive and non recursive functions
17. Read a string (word), store it in an array and obtain its reverse by using a user defined function.
18. Write a menu driven program for performing matrix addition, multiplication and finding the transpose. Use functions to (i) read a matrix, (ii) find the sum of two matrices, (iii) find the product of two matrices, (iv) find the transpose of a matrix and (v) display a matrix.
19. Do the following using pointers
 - i) add two numbers
 - ii) swap two numbers using a user defined function
20. Input and Print the elements of an array using pointers
21. Compute sum of the elements stored in an array using pointers and user defined function.
22. Create a file and perform the following
 - iii) Write data to the file
 - iv) Read the data in a given file & display the file content on console
 - v) append new data and display on console
23. Open a text input file and count number of characters, words and lines in it; and store the results in an output file.

EST 102
PROGRAMMING IN C

MODULE 1

BASICS OF COMPUTER HARDWARE & SOFTWARE

BASICS OF COMPUTER ARCHITECTURE

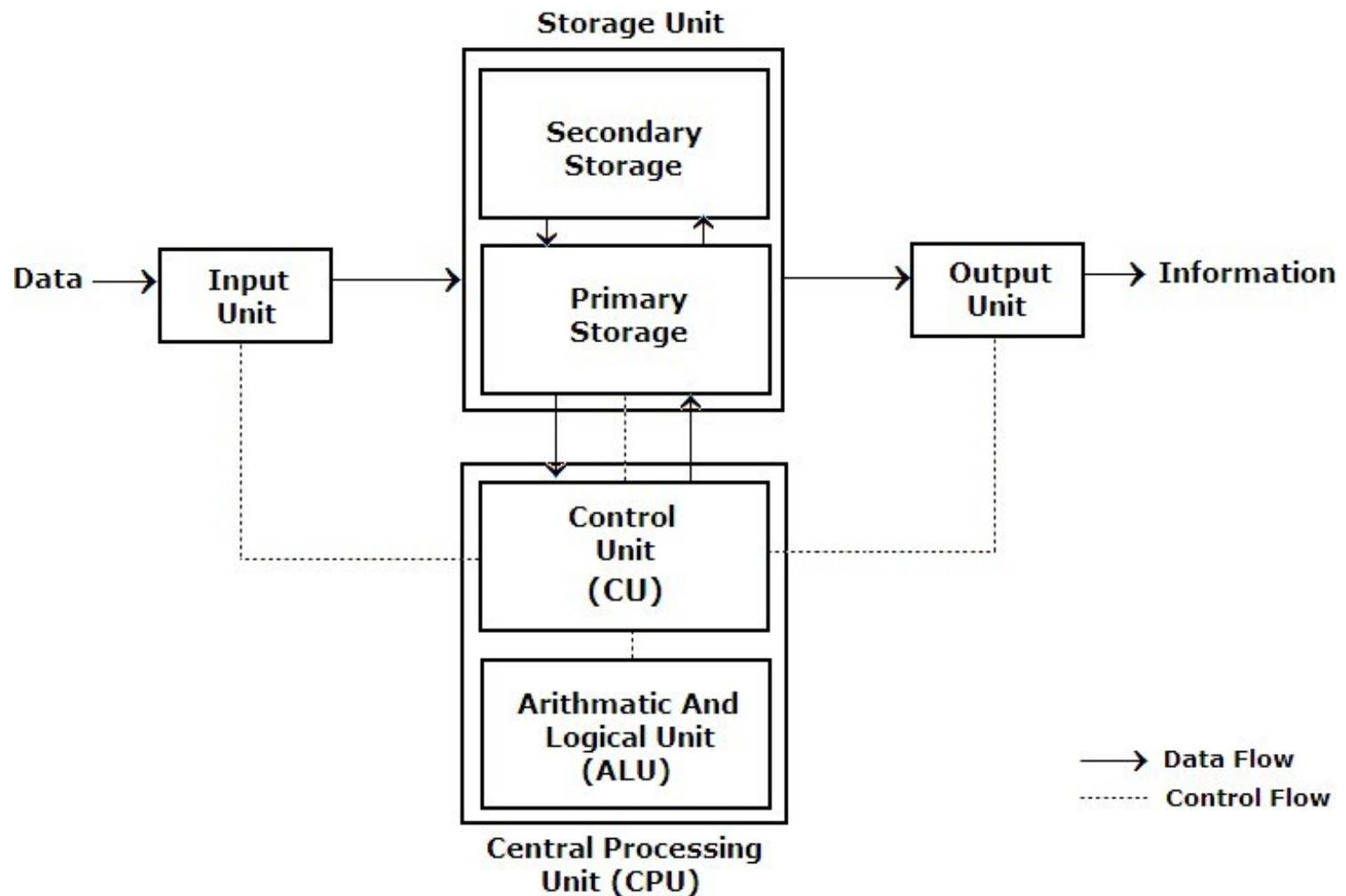
- Computer is an electronic machine that makes performing any task very easy
- In computer, CPU executes each instruction provided to it, in a series of steps, this series of steps is called Machine Cycle, and is repeated for each instruction
- One **machine cycle** involves
 - Fetching of instruction
 - Decoding the instruction
 - Operand fetching

- Executing the instruction

BASICS OF COMPUTER ARCHITECTURE

- Computer system has **five basic units** that help computer to perform operations, which are given below:
 1. Input Unit
 2. Output Unit
 3. Storage Unit
 4. Arithmetic Logic Unit
 5. Control Unit

Basic Units of Computer System



Input Unit

- Input unit connects external environment with internal computer system
- It provides data and instructions to computer system
- Commonly used **input devices** are keyboard, mouse, magnetic tape etc.
- Input unit performs following tasks:
 - Accept data and instructions from outside environment
 - Convert it into machine language
 - Supply the converted data to computer system

Output Unit

- It connects internal system of a computer to external environment
- It provides results of any computation, or instructions to outside world
- Some **output devices** are printers, monitor etc.

Storage Unit

- This unit holds data and instructions
- It also stores intermediate results before these are sent to output devices
- It also stores data for later use
- Storage unit of a computer system can be divided into two categories:
 1. Primary Storage
 2. Secondary Storage

Primary Storage

- Primary memory is computer memory that is accessed directly by CPU
- RAM and ROM is used as primary storage memory
- **RAM(Random Access Memory)** memory is used to store data which is being currently executed
- It is used for temporary storage of data and data is lost, when computer is switched off
- **ROM(Read Only Memory)** Stores crucial information essential to operate system, like program essential to boot computer
- ROM Always retains its data

Secondary Storage

- Every computer also has storage device that's used for storing information on a long-term basis, and is known as secondary storage
- Any file you create or download is saved to computer's secondary storage
- There are two types of storage device used as secondary storage in computers:
 1. HDD (Hard Disk Drive)
 2. SSD (Solid-State Drive)
- HDDs are more traditional of two, SSDs are fast overtaking HDD as preferred one for secondary storage

SSD

- An SSD is a storage medium that uses non-volatile memory to hold and access data
- Unlike a hard drive, an SSD has no moving parts
- Advantages
 1. faster access time
 2. noiseless operation
 3. higher reliability
 4. lower power consumption



HDD

- A hard disk drive (sometimes abbreviated as a hard drive, HD, or HDD) is a non-volatile data storage device
- It is usually installed internally in a computer, attached directly to disk controller of computer's motherboard



Arithmetic Logical Unit (ALU)

- All calculations are performed in ALU of computer system
- ALU can perform basic operations such as addition, subtraction, division, multiplication etc.
- Whenever calculations are required, control unit transfers data from storage unit to ALU
- When the operations are done, result is transferred back to the storage unit

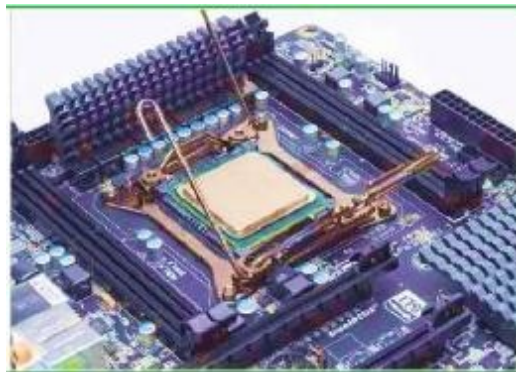
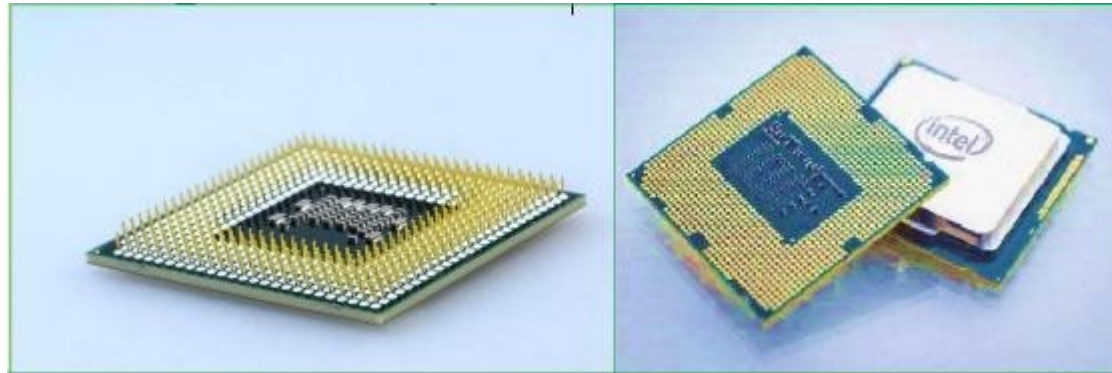
Control Unit (CU)

- It controls all other units of the computer.
- It controls the flow of data and instructions to and from storage unit to ALU
- Thus it is also known as central nervous system of computer

CPU- CENTRAL PROCESSING UNIT

- A processor is an integrated electronic circuit that performs calculations that run a computer
- Known as brain of computer
- A processor performs arithmetical, logical, input/output (I/O) and other basic instructions that are passed from an operating system (OS)
- Most other processes are dependent on operations of a processor
- Terms processor, CPU and microprocessor are commonly linked

CPU



CPU

- Processors can be found in PCs, smartphones, tablets and other computers
- Two main competitors in the processor market are **Intel** and **AMD**
 - It performs following tasks:
 - It performs all operations
 - It takes all decisions
 - It controls all units of computer

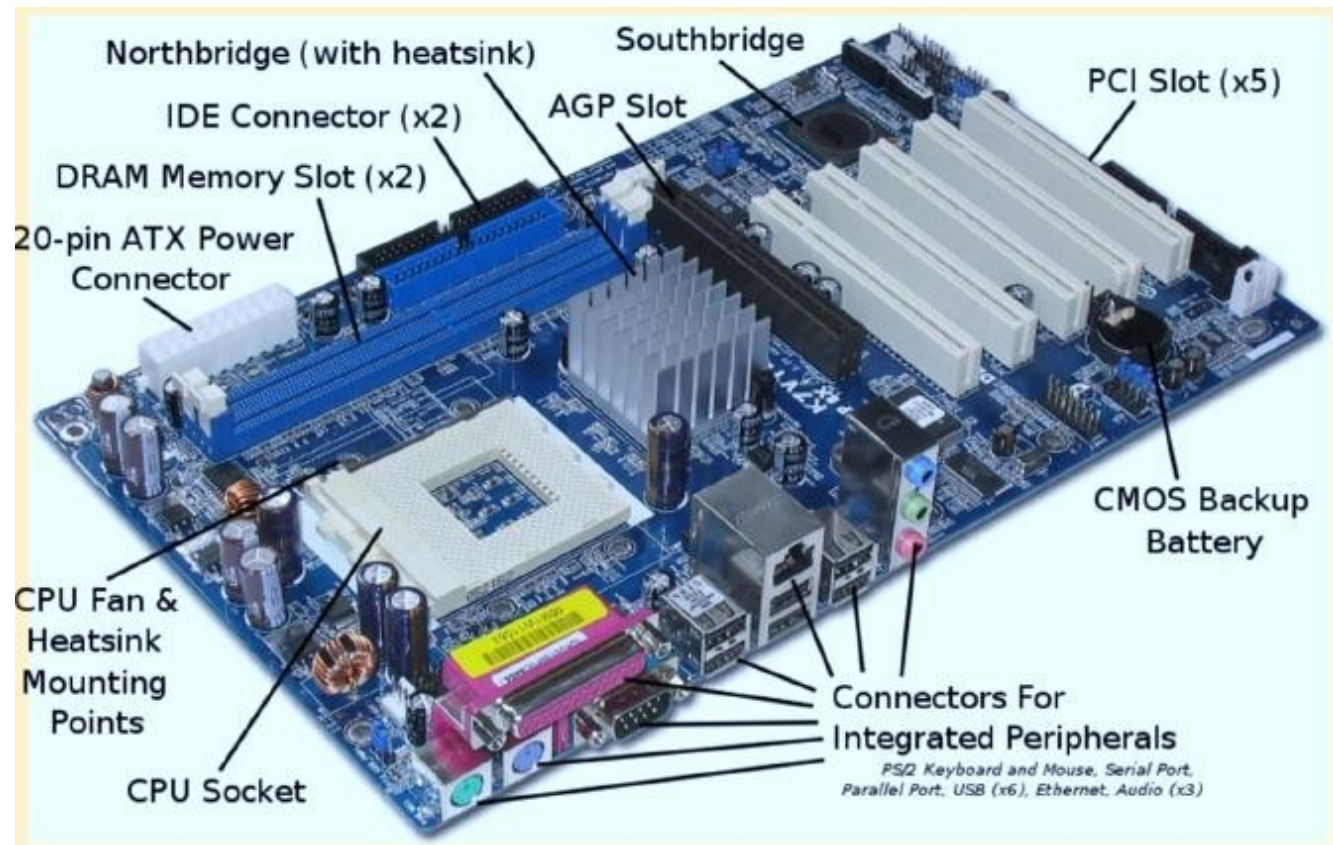
CPU

- Executing a single instruction consists of a particular cycle of events; fetching, decoding, executing and storing
- For example, to do add instruction CPU must
 1. Fetch : get instruction from memory into processor
 2. Decode : internally decode what it has to do(Eg: add)
 3. Execute : take values from registers, actually add them together
 4. Store: store result back into another register

CPU

- Basic elements of a processor include:
 - 1) Arithmetic Logic Unit (ALU)-which carries out arithmetic and logic operations on operands in instructions
 - 2) Floating Point Unit (FPU)-also known as a math coprocessor that is designed to carry out operations on floating-point numbers
 - 3) Registers-which hold instructions and other data. Registers supply operands to ALU and store results of operations
 - 4) Cache memory-Their inclusion in CPU saves time compared to having to get data from RAM

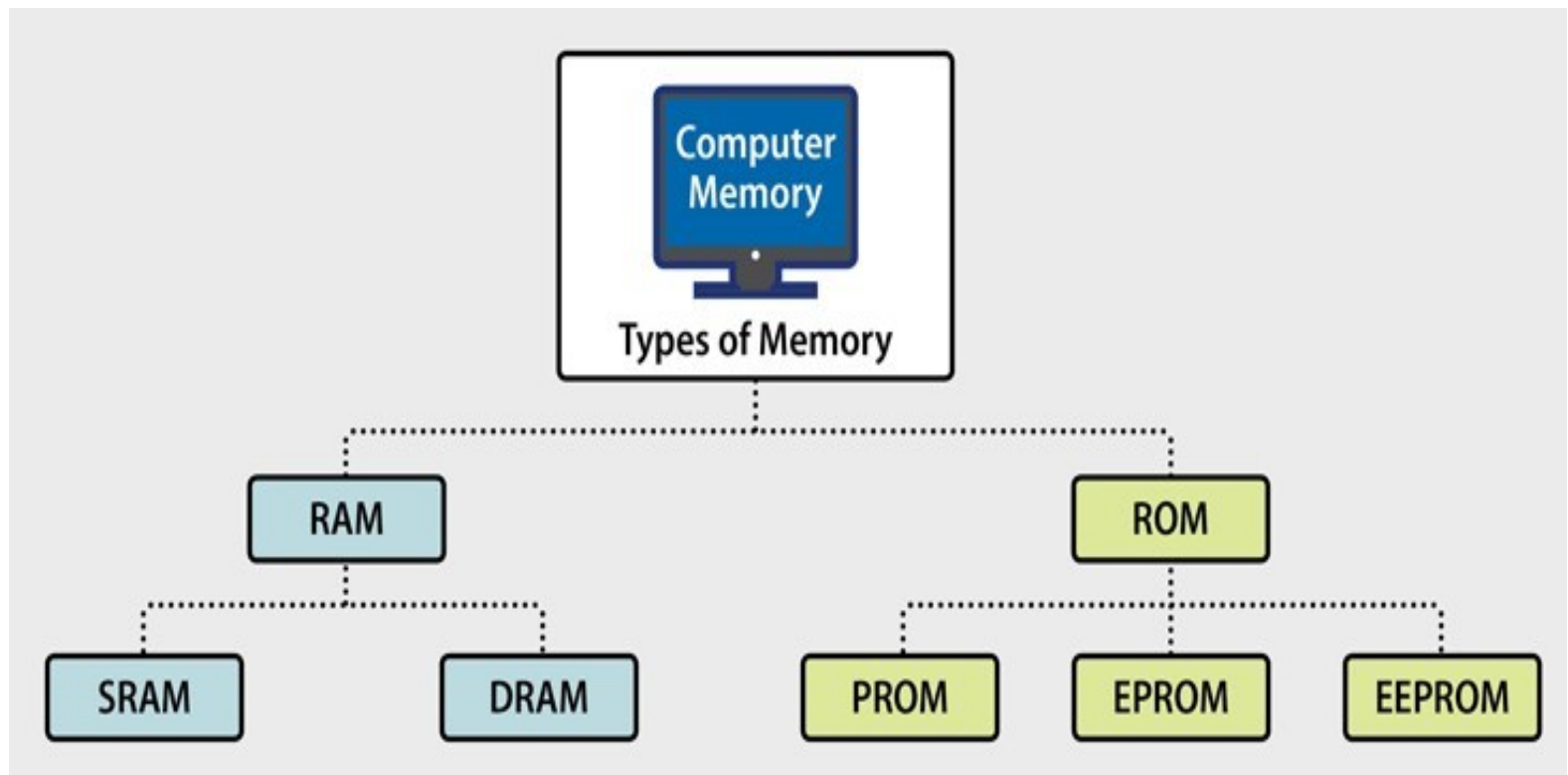
Motherboard



TYPES OF MEMORY

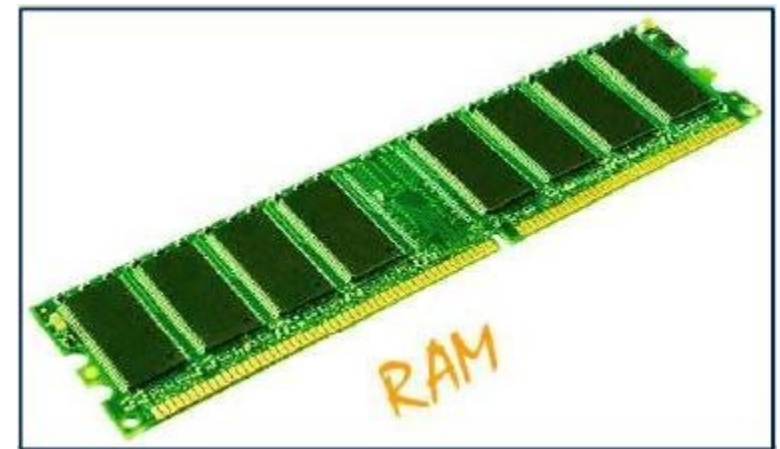
- Memory is the most essential element of a computing system
- Without it computer can't perform simple tasks
- Computer memory is of two basic type - Primary memory(RAM and ROM) and Secondary memory(hard drive, CD, etc.).
- Random Access Memory (RAM) is primary- volatile memory
- Read Only Memory (ROM) is primary-non-volatile memory

TYPES OF MEMORY



RAM - Random Access Memory

- Term RAM is short for Random Access Memory
- RAM is where data computer is working on while computer is running
- Program is also loaded into RAM before being executed
- Both program and data is stored in RAM while program is being executed



RAM

- RAM is typically cleared whenever computer is reset or shutdown
- Thus, data stored in RAM does not survive computer restarts
- Two types
 1. Static RAM(SRAM)
 2. Dynamic RAM(DRAM)

DRAM AND SRAM

DRAM	SRAM
1. Constructed of tiny capacitors that leak electricity.	1.Constructed of circuits similar to D flip-flops.
2.Requires a recharge every few milliseconds to maintain its data.	2.Holds its contents as long as power is available.
3.Inexpensive.	3.Expensive.
4. Slower than SRAM.	4. Faster than DRAM.
5. Can store many bits per chip.	5. Can not store many bits per chip.
6. Uses less power.	6.Uses more power.
7.Generates less heat.	7.Generates more heat.
8. Used for main memory.	8. Used for cache.

Difference between SRAM and DRAM

ROM - Read Only Memory

- Stores crucial information essential to operate system, like program essential to boot computer
- It is not volatile.
- Always retains its data.
- Used in embedded systems
- Used in calculators and peripheral devices
- ROM is further classified into 4 types - ROM, PROM, EPROM, and EEPROM

Types of ROM

- **PROM (Programmable read-only memory)** - It can be programmed by user. Once programmed, data and instructions in it cannot be changed
- **EPROM (Erasable Programmable read only memory)** - It can be reprogrammed. To erase data from it, expose it to ultra violet light. To reprogram it, erase all previous data
- **EEPROM (Electrically erasable programmable read only memory)** - Data can be erased by applying electric field, no need of ultra violet light. We can erase only portions of the chip

INPUT OUTPUT DEVICES

- An input device sends information to a computer system for processing, and an output device reproduces or displays the results of that processing
- Input devices only allow for input of data to computer
- Devices are only input devices or output devices, as they can only accept data input from a user or output data generated by a computer
- Some devices can accept input and display output, and they are referred to as I/O devices (input/output devices)

Input devices

- An input device can send data to another device, but it cannot receive data from another device
- Examples of input devices include following.
 - ▣ **Keyboard and Mouse** - Accepts input from a user and sends that data to computer
 - ▣ **Microphone** - Receives sound generated by an input source, and sends that sound to a computer
 - ▣ **Webcam** - Receives images generated by whatever it is pointed at and sends those images to a computer

Output devices

- An output device can receive data from another device and generate output with that data, but it cannot send data to another device
- Examples of output devices include following
 - **Monitor** - Receives data from a computer and displays that information as text and images for users to view
 - **Projector** - Receives data from a computer and displays, or projects, that information as text and images onto a surface, like a wall or a screen
 - **Speakers** - Receives sound data from a computer and plays sounds for users to hear

Input/output devices

- An input/output device can receive data from a device (input), and send data to another device(output)
- Examples of input/output devices include the following.
- **CD-RW drive and DVD-RW drive** - Receives data from a computer (input), to copy onto a writable CD or DVD. Also, drive sends data contained on a CD or DVD (output) to a computer
- **USB flash drive** - Receives, or saves, data from a computer (input).Also, drive sends data to a computer or another device (output).

SOFTWARES

- **Software** is a collection of instructions that enable user to interact with a computer , its hardware or perform tasks
- Without software, most computers would be useless
- For example, without your Internet **browser software**, you could not surf Internet
- Without an **operating system**, the browser could not run on your computer

SOFTWARES

- There are two types of software
 1. System Software
 2. Application Software
- Examples of **system software** are Operating System, Compilers, Interpreter, Assemblers, etc.
- Examples of **Application software** are Railways Reservation Software, Microsoft Office Suite Software, Microsoft Word, Microsoft PowerPoint , etc.

SOFTWARES



APPLICATION SOFTWARE

- These are basic software used to run to accomplish a particular action and task
- These are dedicated software, dedicated to performing simple and single tasks
- For eg., a single software cannot serve to both reservation system and banking system
- These are divided into two types:
 1. General Purpose Application Software
 2. Specific Purpose Application Software

General Purpose Application Software

- These are types of application software that comes in-built and ready to use, manufactured by some company or someone
- Examples are:
 - Microsoft Excel – Used to prepare excel sheets
 - VLC Media Player – Used to play audio/video files
 - Adobe Photoshop – Used for designing and animation and many more

Specific Purpose Application Software

- These are the type of software that is customizable and mostly used in real-time or business environment
- Examples are:
 - ▣ Ticket Reservation System
 - ▣ Healthcare Management System
 - ▣ Hotel Management System
 - ▣ Payroll Management System

SYSTEM SOFTWARES

- Systems software includes programs that are dedicated to managing computer itself, such as the operating system and disk operating system (or DOS)
- System software is a software that provides platform to other software's
- Some examples can be operating systems, antivirus software, disk formatting software, Computer language translators etc.

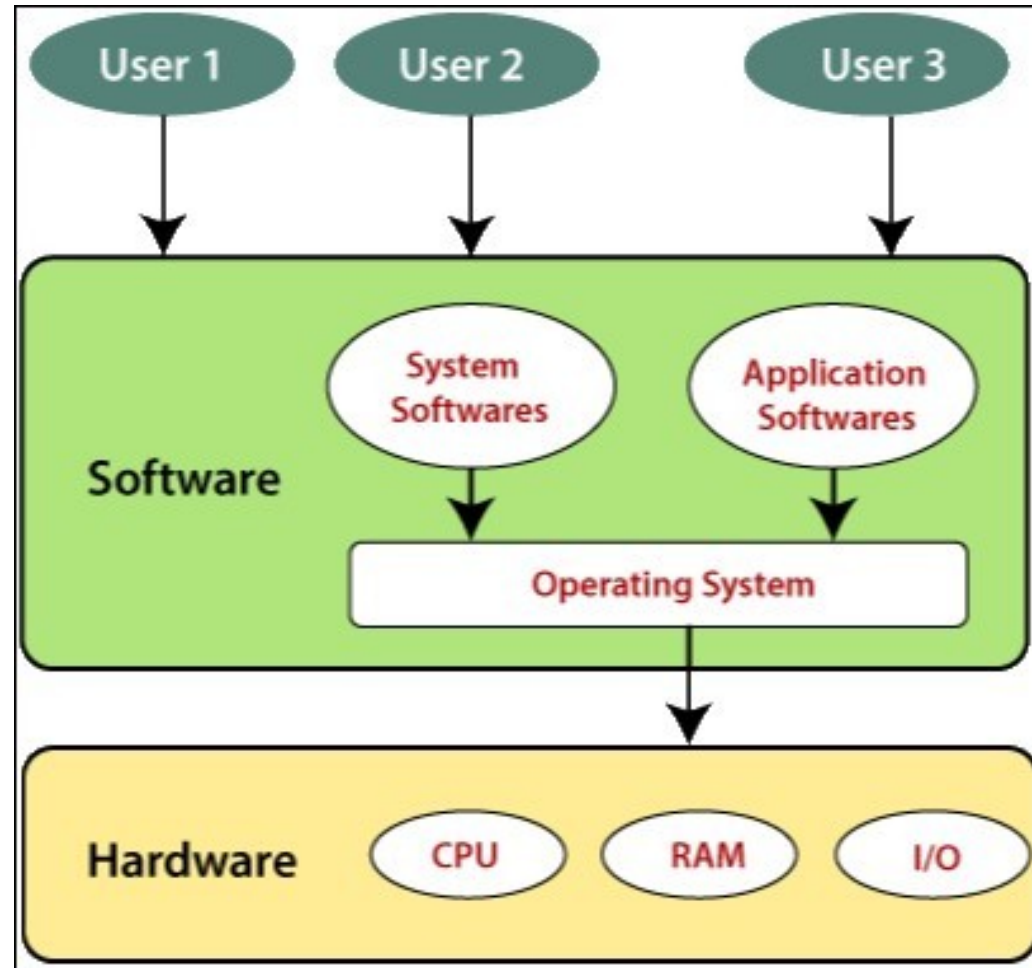
SYSTEM SOFTWARES

- These are commonly prepared by computer manufacturers
- These software's consists of programs written in low-level languages, used to interact with hardware at a very basic level
- System software serves as interface between hardware and end users
- Important features of system software include:
 - 1. Closeness to system
 - 2. Fast speed
 - 3. Difficult to manipulate and design
 - 4. Written in low level language

OPERATING SYSTEM

- An operating system (OS) is a type of system software that manages computer's hardware and software resources
- It provides common services for computer programs
- OS acts a link between software and hardware
- It controls and keeps a record of execution of all other programs that are present in computer, including application programs and other system software

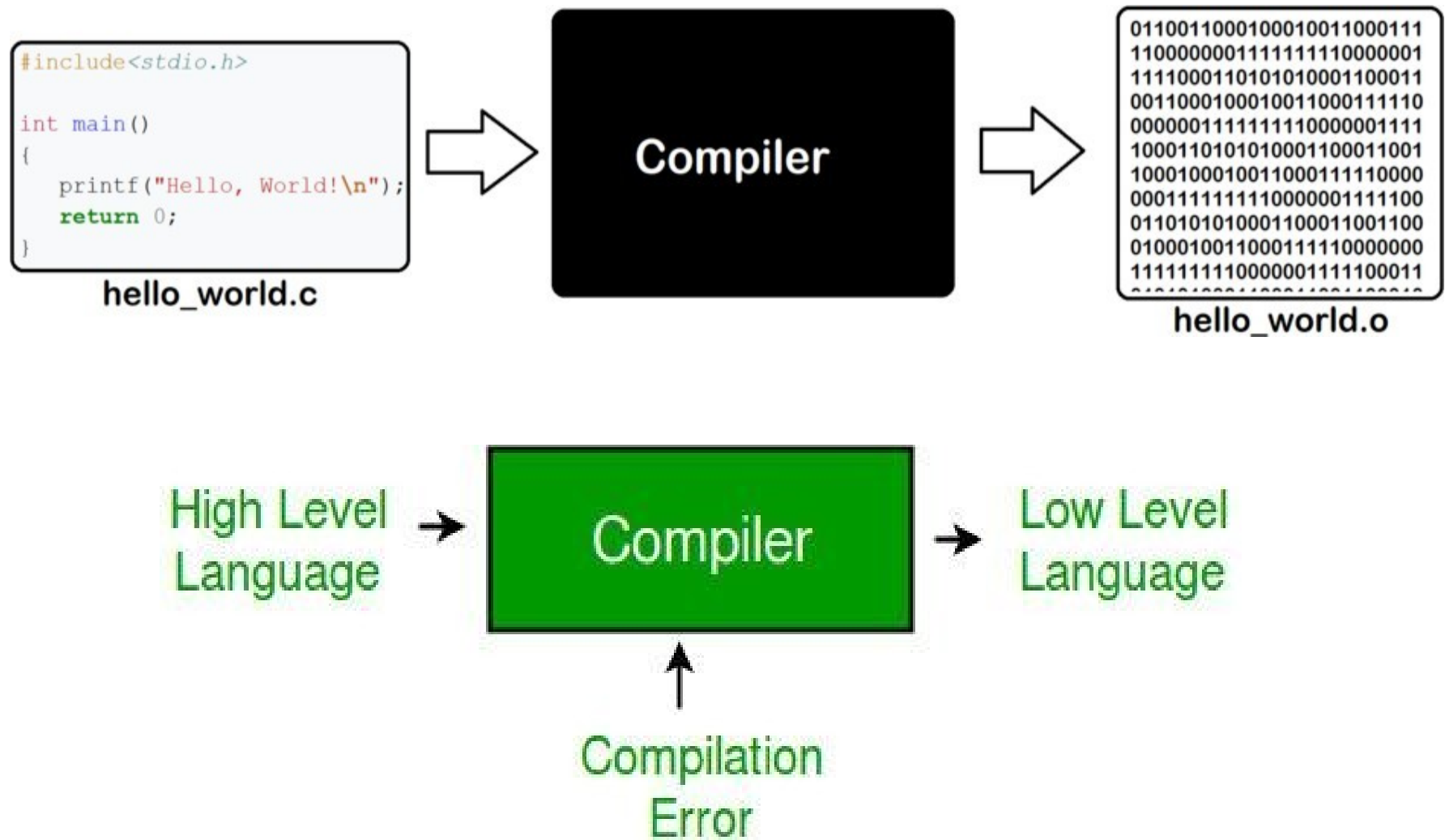
OPERATING SYSTEM



COMPILER

- A compiler is a computer program which transforms source code written in a high-level language into low-level machine language
- A compiler is a software that translates code written in one programming language (source language) to another language (target language) without changing meaning of program
- Compiler is also said to make target code efficient and optimized in terms of time and space
- Examples of compiler may include gcc (C compiler), g++ (C++ Compiler), javac (Java Compiler) etc.

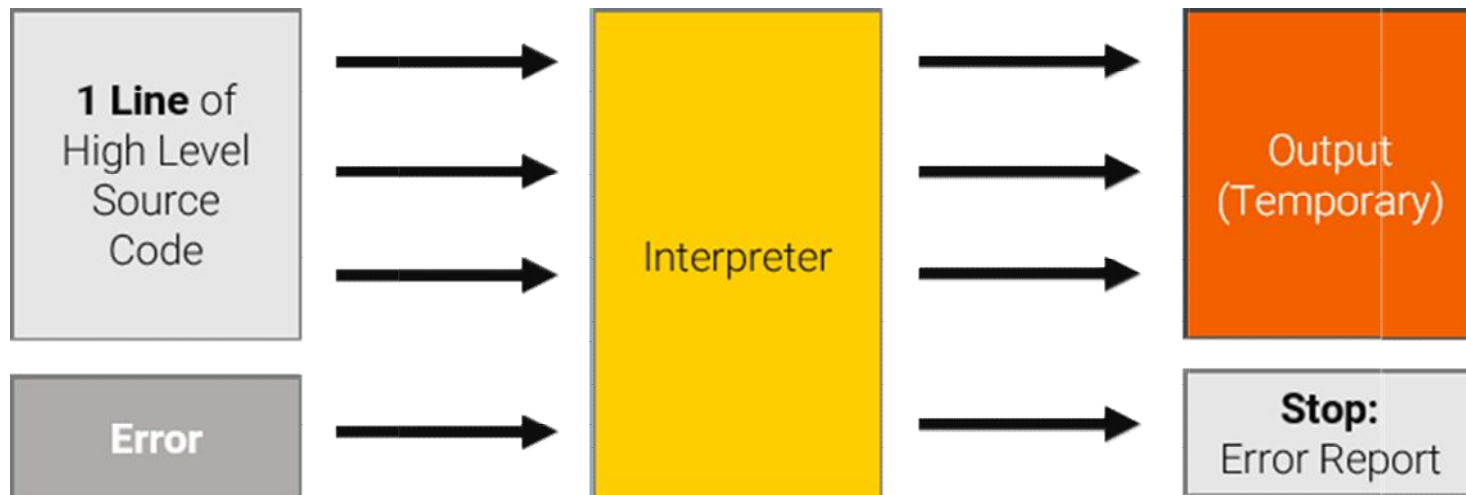
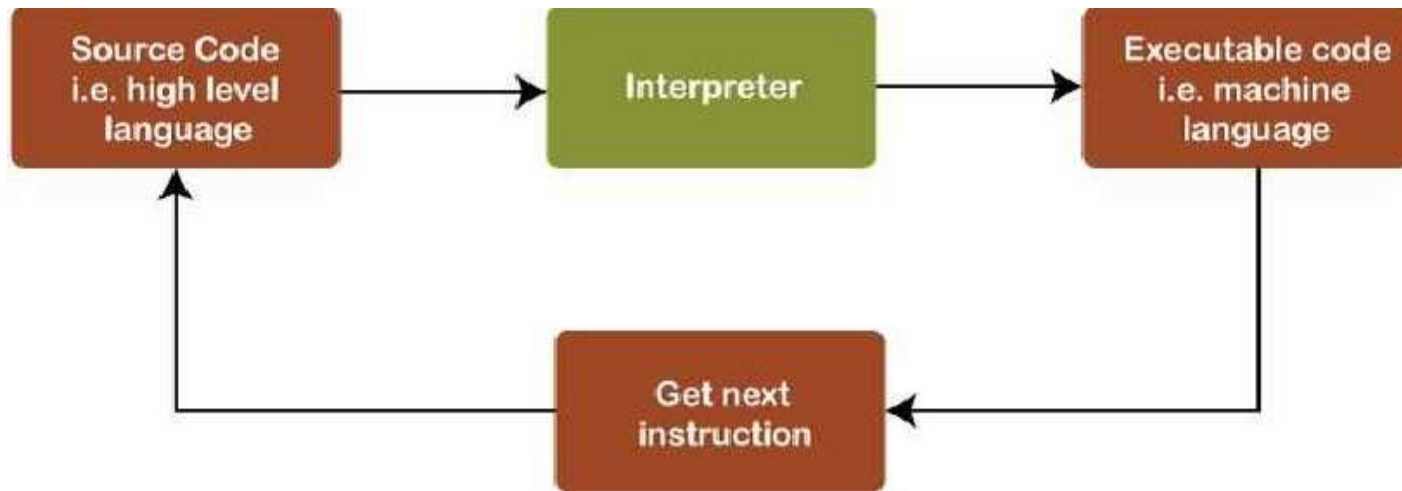
COMPILER



INTERPRETER

- An interpreter, like a compiler, translates high-level language into low-level machine language
- A compiler reads whole source code at once and generates machine code
- An interpreter reads a statement or one line from source code, converts it to a machine code, then takes next statement in sequence
- If an error occurs, an interpreter stops execution and reports it
- A compiler reads whole program even if it encounters several errors
- Examples may include Ruby, Python, PHP etc.

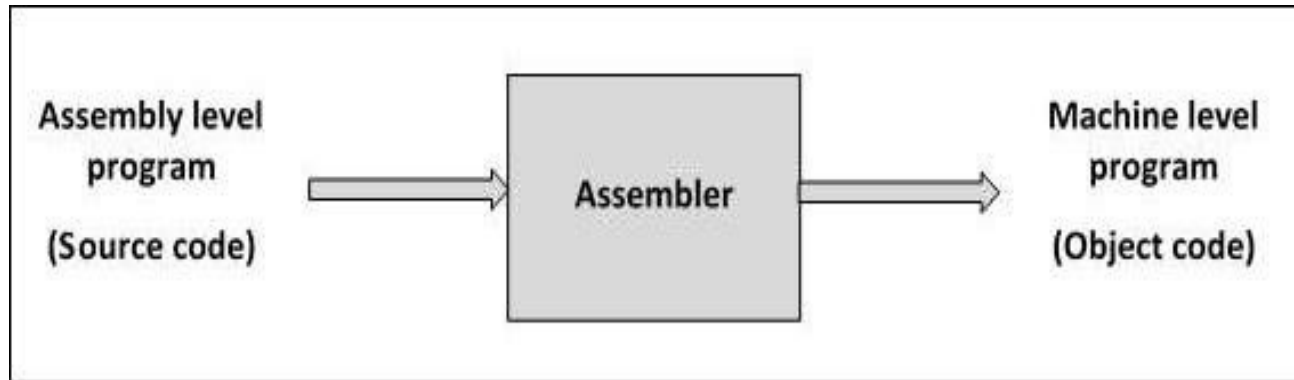
INTERPRETER



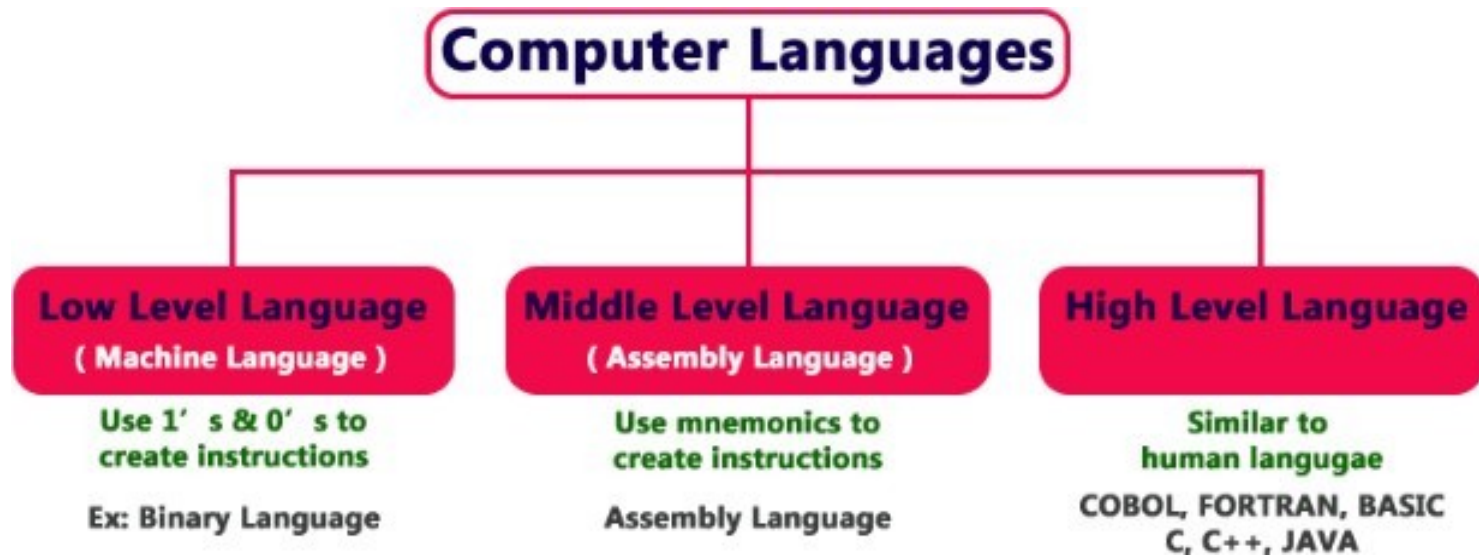
ASSEMBLER

- An assembler is a program that converts assembly language into machine code
- Assemblers produce executable code that similar to compilers
- However, assemblers are more simplistic since they only convert low-level code (assemblylanguage) to machine code

ASSEMBLER



COMPUTER LANGUAGES



Low-level language

- It is represented in 0 or 1 forms, which are machine instructions
- Languages that come under this category are Machine level language and Assembly language

Machine-level language

- Machine-level language is a language that consists of a set of instructions that are in binary form 0 or 1
- Computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1
- Instructions given to computer can be only in binary code
- Creating a program in a machine-level language is a very difficult task as it is not easy for programmers to write program in machine instructions

Machine-level language

- It is error-prone as it is not easy to understand, and its maintenance is also very high
- A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer will no longer be valid in another computer

Assembly Language

- Assembly language contains some human-readable commands such as mov, add, sub, etc.
- Problems which we were facing in machine-level language are reduced to some extent by using an extended form of machine-level language known as assembly language
- Since assembly language instructions are written in English words like mov, add, sub, so it is easier to write and understand

Assembly Language

- As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code
- Translator used for translating the code is known as an assembler
- Assembly language code is not portable
- Assembly code is not faster than machine code because assembly language comes above the machine language in hierarchy

High-Level Language

- High-level language is a programming language that allows a programmer to write programs which are independent of a particular type of computer
- High-level languages are considered as high-level because they are closer to human languages than machine-level languages
- A compiler is required to translate a high-level language into a low-level language
- Examples of high level languages are C, C++, Java, Python, etc.

Advantages of high-level languages

- High-level language programs are easy to get developed
- It is easy to visualize function of program
- Programmer may not remain aware about architecture of hardware
- So people without hardware knowledge can also do high level language programming.
- Same high level language program may works on different computers, so high-level languages are portable

Disadvantages of high-level languages

- A high level language program can't get executed directly
- It requires some translator to get it translated to machine language
- There are two types of translators for high level language programs. They are interpreter and compiler
- These translator programs, especially compilers, are huge one and so are quite expensive

Difference Between High Level and Low Level Languages

S.NO	HIGH LEVEL LANGUAGE	LOW LEVEL LANGUAGE
1.	It is programmer friendly language.	It is a machine friendly language.
2.	High level language is less memory efficient.	Low level language is high memory efficient.
3.	It is easy to understand.	It is tough to understand.
4.	It is simple to debug.	It is complex to debug comparatively.
5.	It is simple to maintain.	It is complex to maintain comparatively.
6.	It is portable.	It is non-portable.
7.	It can run on any platform.	It is machine-dependent.
8.	It needs compiler or interpreter for translation.	It needs assembler for translation.

STRUCTURED PROGRAMMING

- A programming approach in which program is made as a single structure
- Also called as **modular programming**
- It means that code will execute instruction by instruction one after other
- It doesn't support possibility of jumping from one instruction to some other with help of any statement like GOTO, etc.
- Therefore, instructions in this approach will be executed in a serial and structured manner
- Languages that support Structured programming approach are: C, C++, JAVA, C# etc

STRUCTURED PROGRAMMING

- Structured program mainly consists of three types of elements:
 - ▢ Selection Statements
 - ▢ Sequence Statements
 - ▢ Iteration Statements
- Structured program consists of well-structured and separated modules
- But entry and exit in a structured program is a single time event

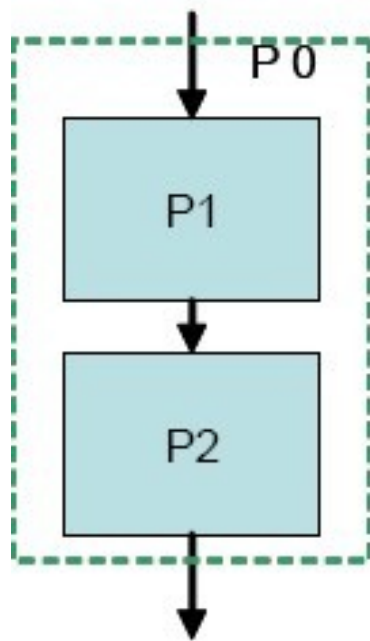
STRUCTURED PROGRAMMING

- Structured programming is a type of programming that involves **breaking the program into smaller modules of code**
- Modules have a duty of performing a single task
- It means that program uses single-entry and single-exit elements
- Therefore a structured program is well maintained, neat and clean program
- This is reason why the Structured Programming Approach is well accepted in programming world
- Mechanisms that allow us to control flow of execution are called control structures

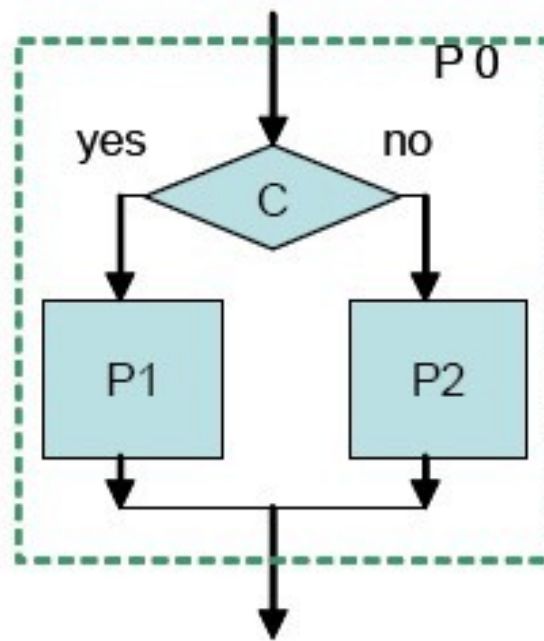
STRUCTURED PROGRAMMING

- There are three main categories of control structures
 - **Sequence** - Simply do one instruction then next and next.
 - **Selection or decision** - This is where you select or choose between two or more flows. Choice is decided by asking some sort of question. Answer determines which lines of code will be executed
 - **Iteration or loop** - Also known as repetition, it allows some code to be executed (or repeated) several times

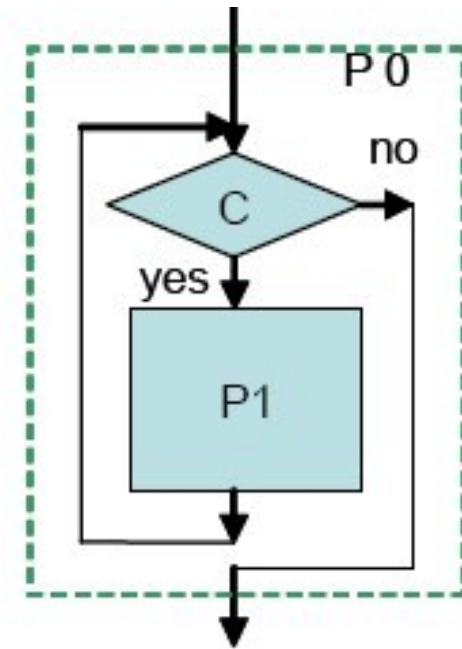
STRUCTURED PROGRAMMING



(a) Sequencing



(b) Selection



(c) Loop

Advantages of Structured Programming Approach

- Easier to read and understand
- User Friendly
- Easier to Maintain
- Mainly problem based instead of being machine based
- Development is easier as it requires less effort and time
- Easier to Debug
- Machine-Independent, mostly

Disadvantages of Structured Programming Approach

- Since it is Machine-Independent, So it takes time to convert into machine code.
- Converted machine code is not same as for assembly language
- Program depends upon changeable factors like data-types
- Therefore it needs to be updated with need on go
- Usually development in this approach takes long time as it is language-dependent
- In case of assembly language, development takes lesser time as it is fixed for machine

ALGORITHM & FLOWCHART

- **Algorithm** is list of instructions and rules that a computer needs to do to complete a task
- Algorithm is a step by step procedure, to solve a problem
- **Flow chart** is a graphical representation of an algorithm
- It is a program-planning tool to solve a problem
- It makes use of symbols which are connected among them to indicate flow of information and processing
- Process of drawing a flowchart for an algorithm is known as **flowcharting**







ALGORITHM & FLOWCHART

No.	Algorithm	Flowchart
1.	Algorithm is step by step procedure to solve the problem.	Flowchart is a diagram created by different shapes to show the flow of data.
2.	Algorithm is complex to understand.	Flowchart is easy to understand.
3.	In algorithm plain text are used.	In flowchart, symbols/shapes are used.
4.	Algorithm is easy to debug.	Flowchart it is hard to debug.
5.	Algorithm is difficult to construct.	Flowchart is simple to construct.
6.	Algorithm does not follow any rules.	Flowchart follows rules to be constructed.

ALGORITHM & FLOWCHART

- Algorithm has following characteristics
 - Input: An algorithm may or may not require input
 - Output: Each algorithm is expected to produce at least one result
 - Definiteness: Each instruction must be clear and unambiguous
 - Finiteness: Algorithm should terminate after finite number of steps
- Flowchart is diagrammatic/Graphical representation of sequence of steps to solve a problem

FLOWCHART SYMBOLS

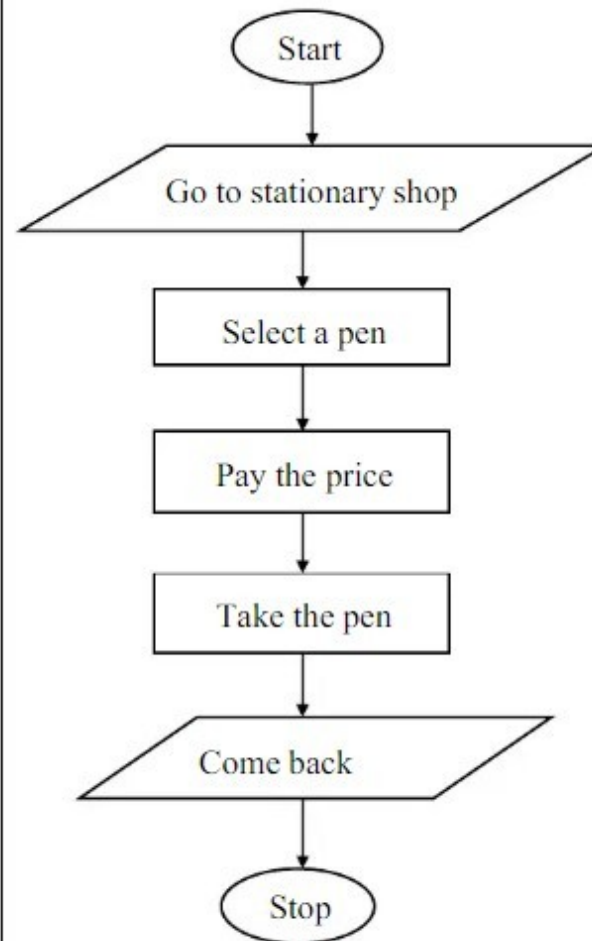
Symbol	Symbol Name
	Terminal
	Process
	Decision
	Input / Output
	Connector
	Flow line

Algorithm and flowchart to Buy a Pen

Algorithm

- Step1. Start.
- Step2. Go to stationary shop
- Step3. Select a pen
- Step4. Pay the price
- Step5. Take the pen
- Step6. Come back home
- Step7. Stop

Flowchart

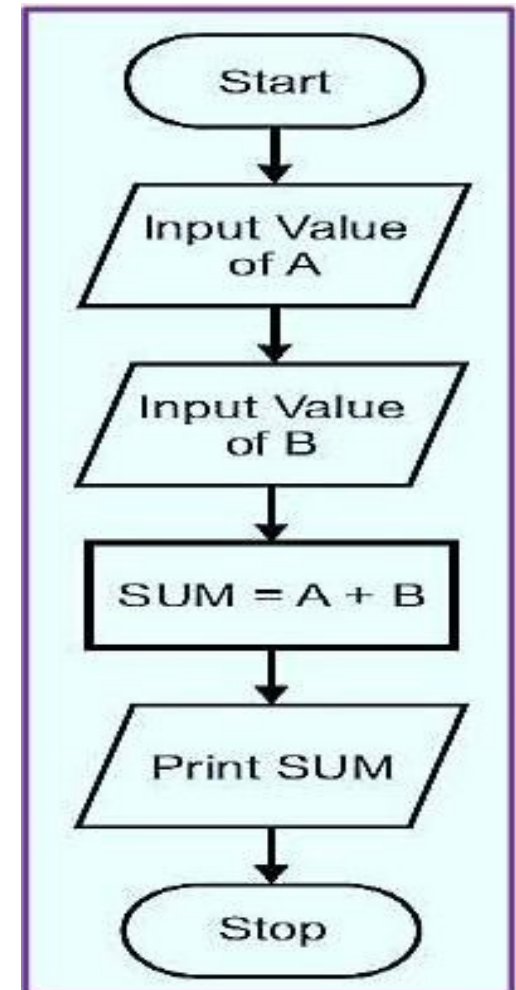


Algorithm & Flowchart to find sum of two numbers

Algorithm

- Step-1 Start
- Step-2 Input first number say A
- Step-3 Input second number say B
- Step-4 $SUM = A + B$
- Step-5 Display SUM
- Step-6 Stop

Flowchart



Algorithm & Flowchart to find Area and Perimeter of Circle

Algorithm

Step-1 Start

Step-2 Input Radius of Circle say R

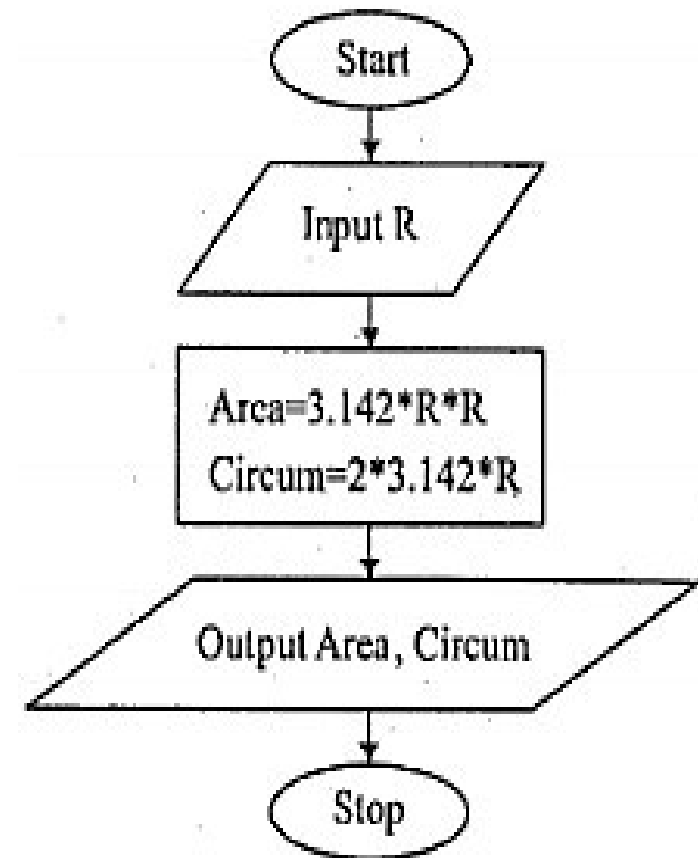
Step-3 $AREA = 3.142 * R * R$

Step-4 $CIRCUM = 2 * 3.142 * R$

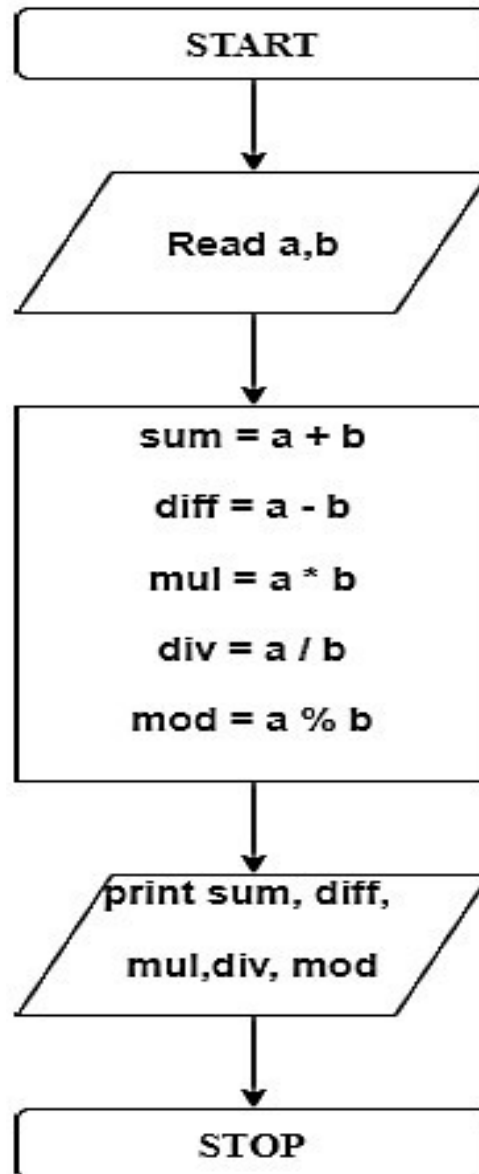
Step-5 Display AREA,

CIRCUM

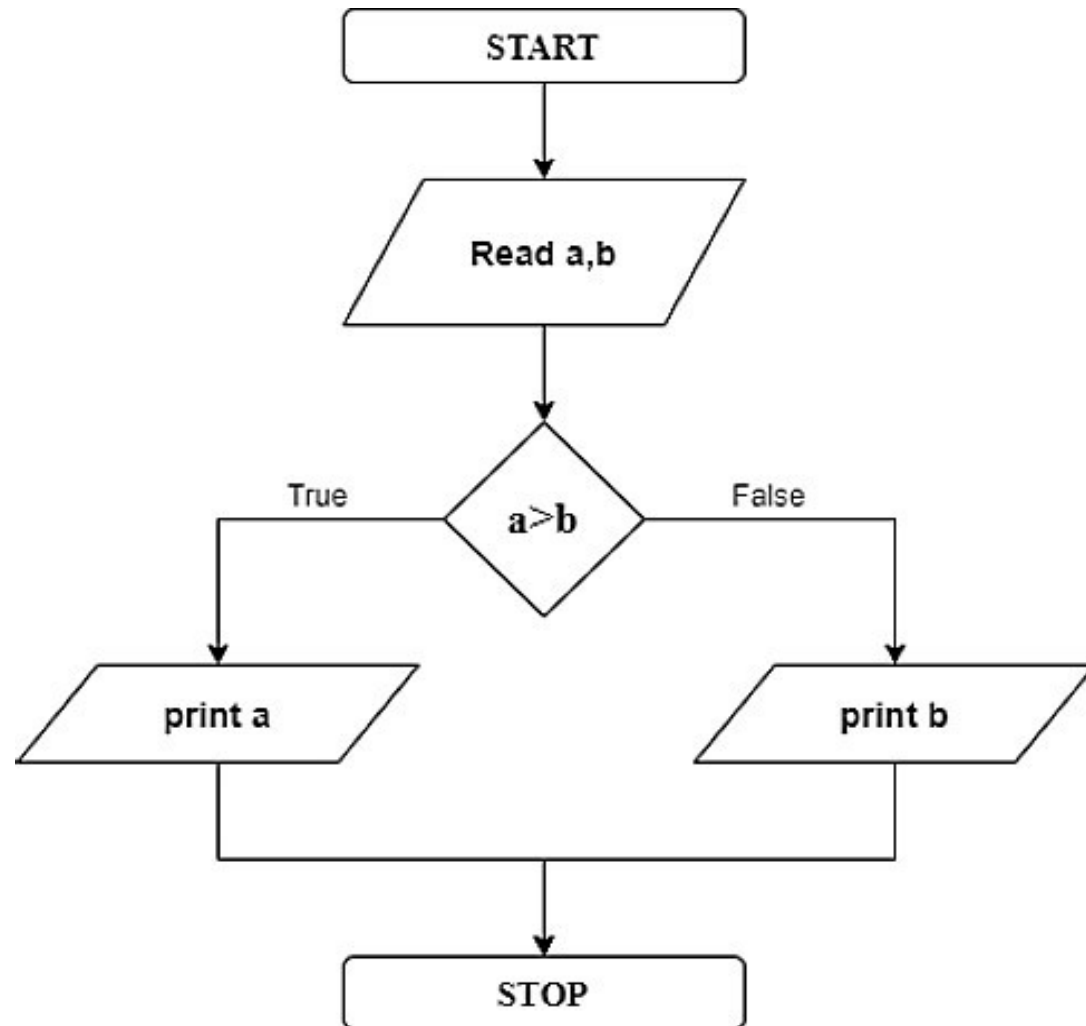
Step-6 Stop



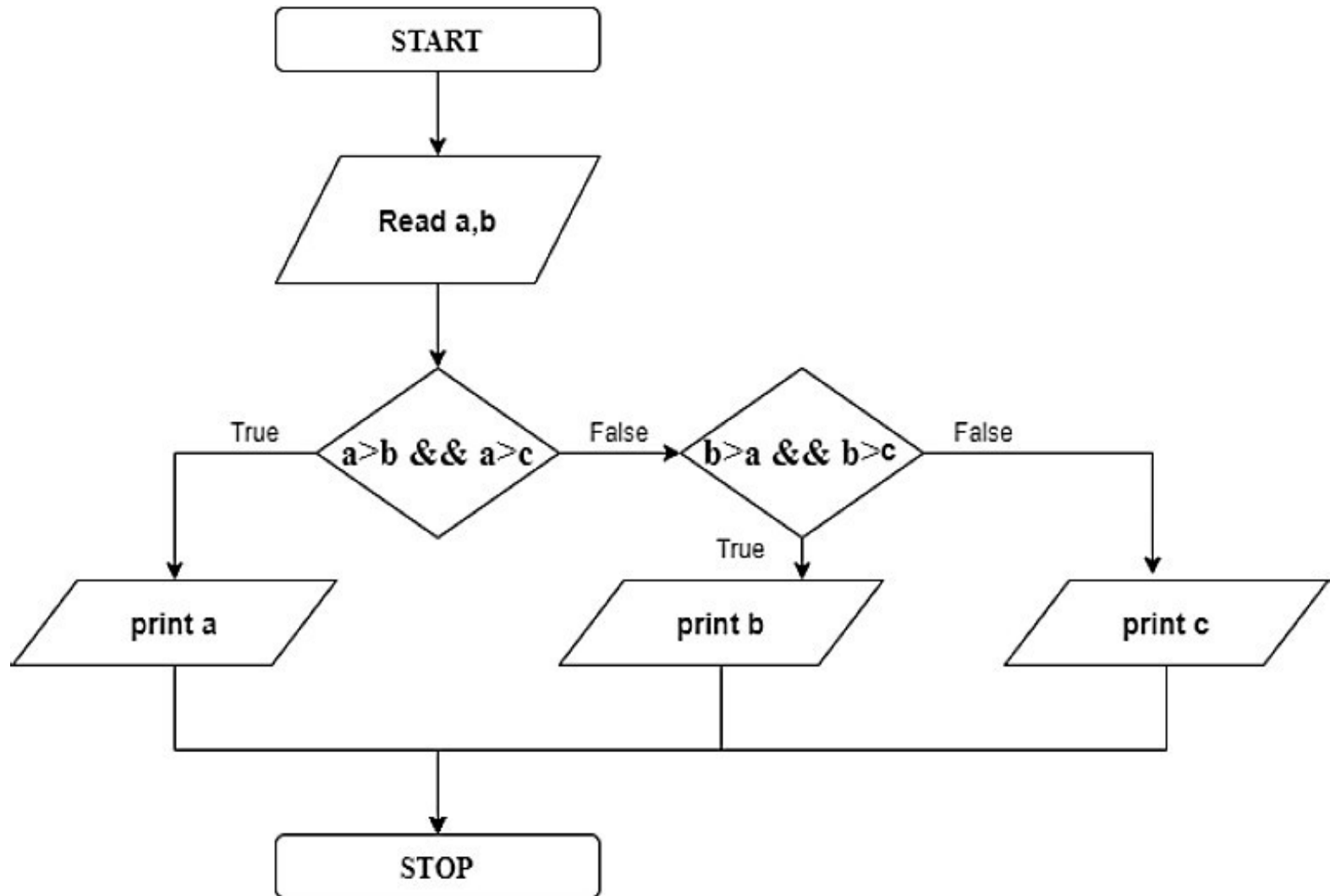
Flowchart for Arithmetic Operation



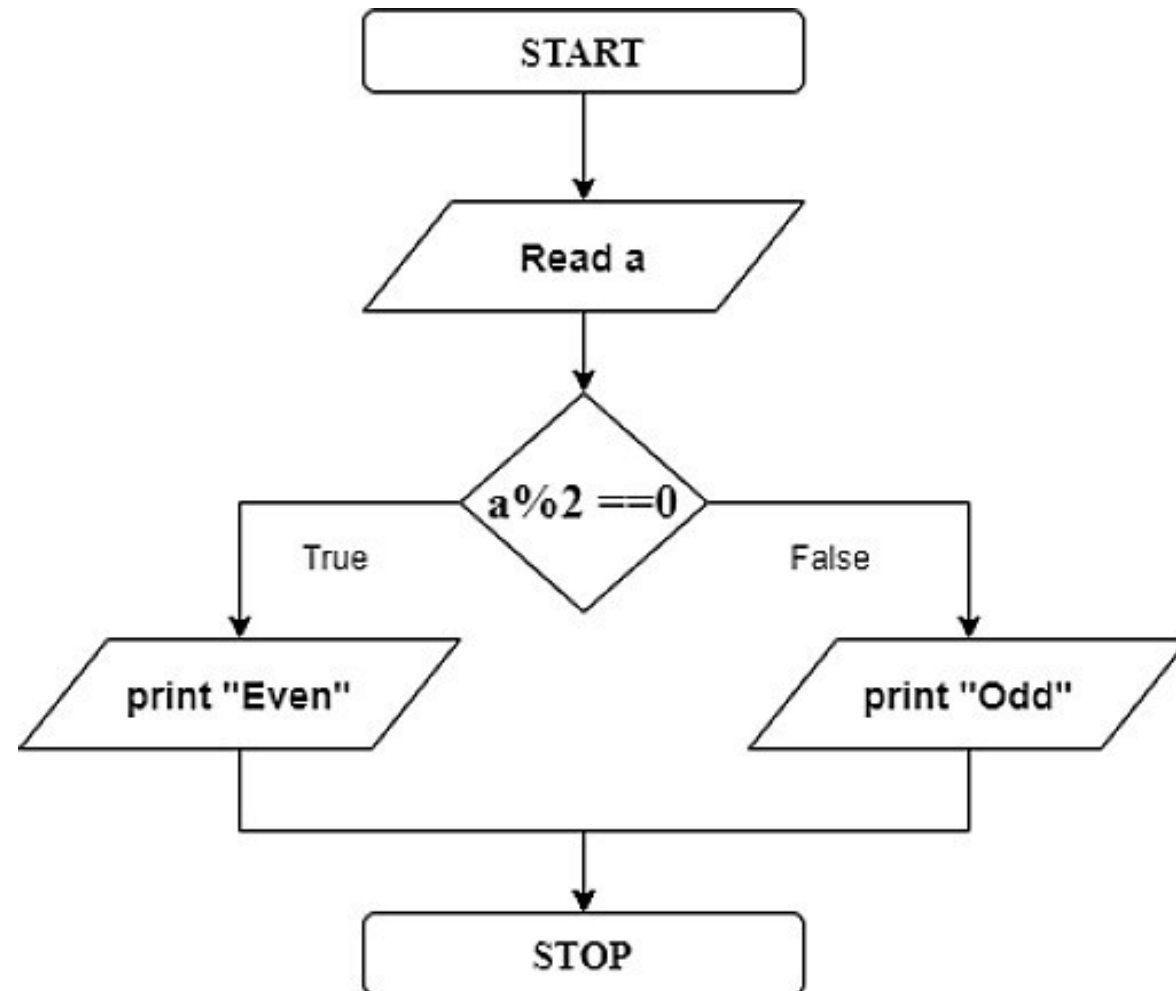
Flowchart to find Largest of two numbers



Flowchart to find Largest of three numbers



Flowchart to check whether a number is odd or even



PSEUDO CODE

- It's simply an implementation of an algorithm in form of annotations and informative text written in plain English
- It has no syntax like any of programming language and thus can't be compiled or interpreted by computer
- It's the cooked up representation of an algorithm
- Pseudo code, as name suggests, is a false code or a representation of code which can be understood by person with some school level programming knowledge

Advantages of Pseudo code

- Improves **readability** of any approach. It's one of best approaches to start implementation of an algorithm
- Acts as a **bridge** between program and algorithm or flowchart
- Also works as a rough **documentation**, so program of one developer can be understood easily when a pseudo code is written out
- Easier **bug(error) detection** and fixing

Advantages of Pseudo code

- In industries, approach of documentation is essential. And that's where a pseudo-code proves vital
- Main goal of a pseudo code is to explain what exactly each line of a program should do, hence making **code construction** phase easier for programmer

Disadvantages of Pseudo code

- Pseudo code does not provide a visual representation of logic of programming
- There are no proper format for writing pseudo code
- In Pseudo code there is extra need of maintain documentation
- In Pseudo code there is no proper standard, many companies follow their own standard for writing the pseudo code

ALGORITHM V/S PSEUDOCODE

- **Algorithm:-**

1. Start
2. Read Values a and b
3. Compute $a + b$ and store result in sum.
4. Print value of Sum
5. Stop

- **Pseudo Code:-**

Begin

input a, b

sum = $a + b$

Print sum

End

ALGORITHM V/S PSEUDOCODE

Problem: Swap the values inside two integer variables. For example, if $a = 7$ and $b = 5$, then make $a = 5$ and $b = 7$.

Algorithm

Step 1: Let $a = 7$ and $b = 5$

Step 2: Create a temporary variable called c

Step 3: Assign c the value of a

Step 4: Assign a the value of b

Step 5: Assign b the value of c

Step 6: Now, the value of a is 5 and b is now 7

Pseudocode

set $a = 7$

set $b = 5$

$c = a;$

$a = b;$

$b = c;$

LINEAR SEARCH

- Linear search is simplest search algorithm and often called **sequential search**
- In this type of searching, we simply traverse list completely and match each element of list with item whose location is to be found
- If match found then location of item is returned otherwise algorithm return NULL

LINEAR SEARCH

5	4	3	2	1
---	---	---	---	---

searching for 3

5	4	3	2	1
---	---	---	---	---

5 === 3? No, next!

5	4	3	2	1
---	---	---	---	---

4 === 3? No, next!

5	4	3	2	1
---	---	---	---	---

3 === 3? Yes, found it!

LINEAR SEARCH EXAMPLE

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

LINEAR SEARCH EXAMPLE

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

LINEAR SEARCH EXAMPLE

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

ALGORITHM – LINEAR SEARCH

Linear Search (Array A, Value x)

Step 1: Set i to 0

Step 2: if $i = n$ then go to step 7 Step 3:

if $A[i] = x$ then go to step 6 Step 4: Set

i to $i+1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found Step 8:

Stop

PSEUDOCODE – LINEAR SEARCH

Begin

for $i = 0$ to $(n - 1)$ **do****if**

$(A[i] = x)$ **then**

 Print x Found at index i

 Exit

end if

else

$i = i + 1$

endfor

 Print x not Found”

End

BUBBLE SORT

- Sorting refers to ordering data in an increasing or decreasing fashion according to some linear relationship among data items
- Bubble sort is a simple **sorting algorithm**
- This sorting algorithm is comparison-based algorithm in which **each pair of adjacent elements is compared** and the elements are swapped if they are not in order
- This algorithm is **not suitable for large data sets** as its average and worst case complexity are of $O(n^2)$ where n is the number of items

BUBBLE SORT

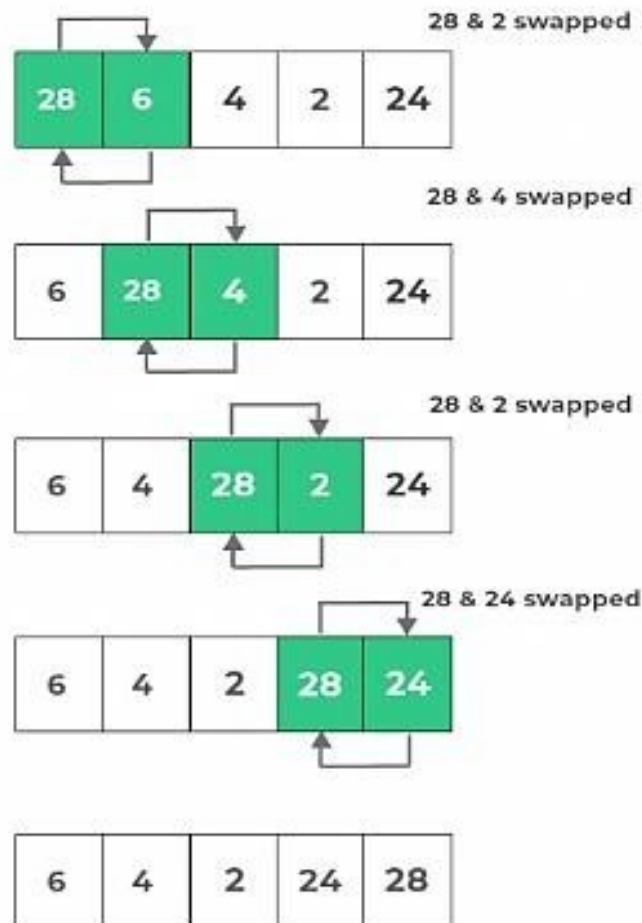
- If given array has to be sorted in **ascending order**, then bubble sort will start by comparing first element of array with second element
- If first element is greater than second element, it will **swap** both elements, and then move on to compare second and the third element, and so on
- If we have total **n elements**, then we need to repeat this process for **n-1 times**
- It is known as bubble sort, because with every complete iteration largest element in given array, bubbles up towards last place or highest index, just like a water bubble rises up to water surface

BUBBLE SORT EXAMPLE

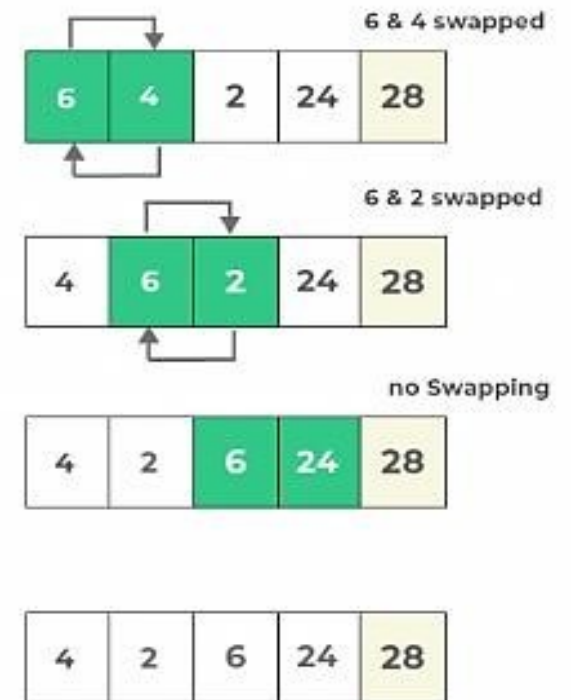
List

28	6	4	2	24
----	---	---	---	----

Pass 1

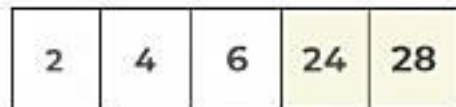
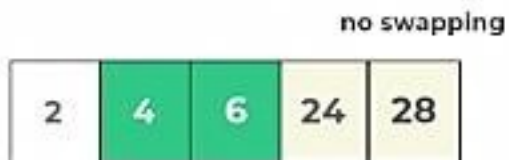
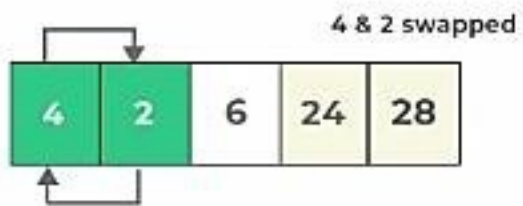


Pass 2

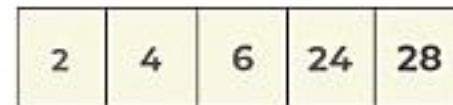
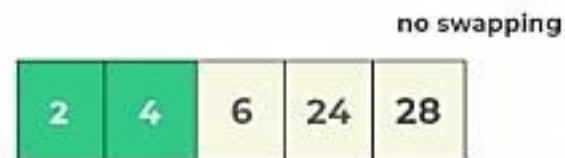


BUBBLE SORT EXAMPLE

Pass 3



Pass 4



Final Result



BUBBLE SORT - ALGORITHM

- Step 1: Start
- Step 2: In Pass 1, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$, $A[2]$ is compared with $A[3]$ and so on; elements are swapped if they are not in order. At end of pass 1, largest element of list is placed at highest index of list.
- Step 3: In Pass 2, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$ and so on; elements are swapped if they are not in order. At end of Pass 2 second largest element of list is placed at second highest index of list.
- Step 4: In pass $n-1$, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$ and so on; elements are swapped if they are not in order. At end of this pass, smallest element of list is placed at first index of list.
- Step 5: Stop

BUBBLE SORT - PSEUDOCODE

```
begin Bubble_Sort(list)
  for all elements of list
    if list[i]>list[i+1]
      swap(list[i], list[i+1])
    end if
  end for
  return list
end Bubble_Sort
```

END....

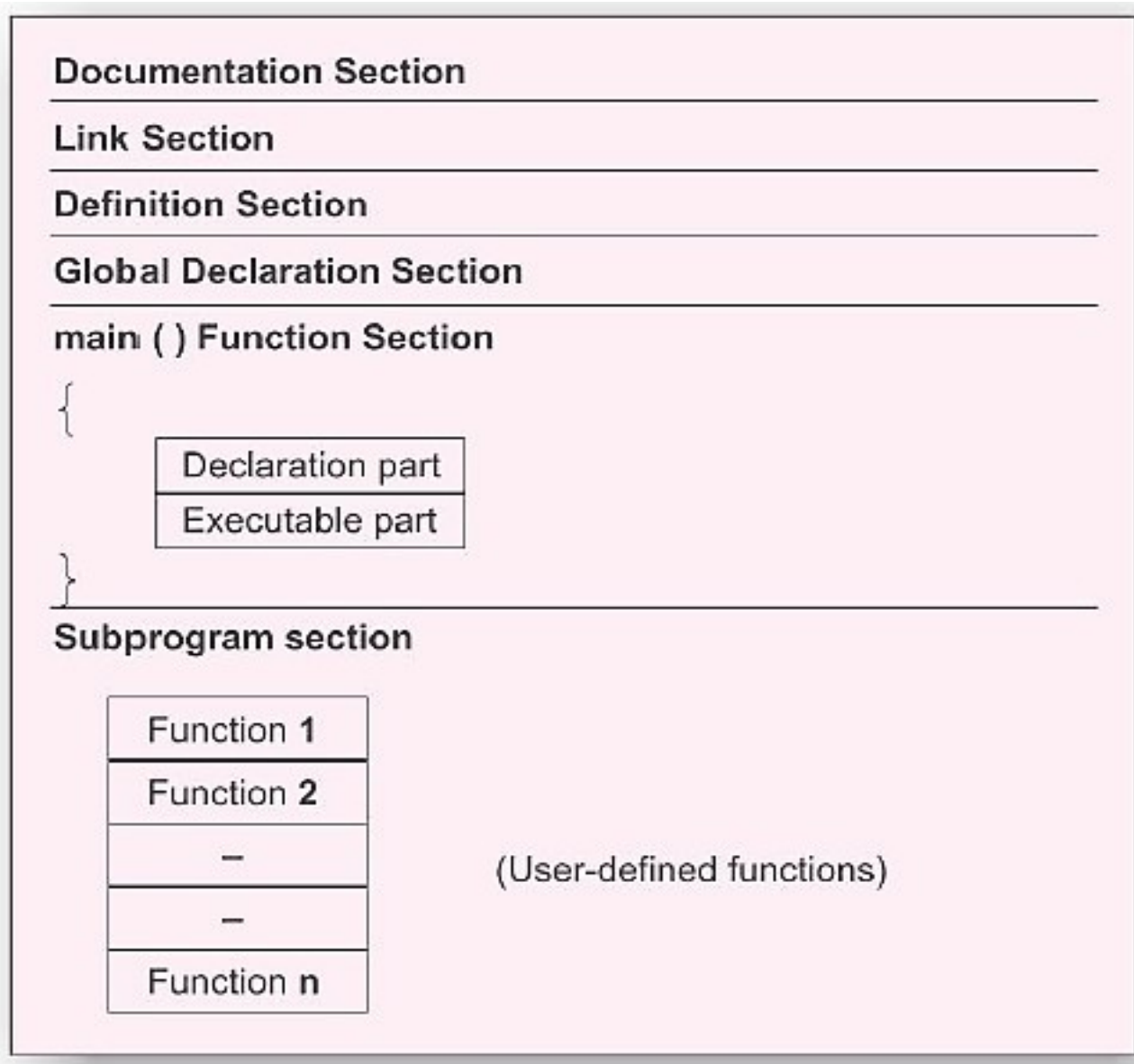
MODULE 2

BASICS OF C PROGRAMMING

C PROGRAM

- Set of instructions that are provided to computer is known as a **program** and development of program is known as **programming**
- Programming is a problem-solving activity
- C was evolved from ALGOL, BCPL (Basic Combined Programming Language) and B by **Dennis Ritchie**
- C is **structured, high level, machine independent** language
- C is a very powerful and widely used language
- It forms (or is basis for) core of modern languages Java and C++

STRUCTURE OF A C PROGRAM



STRUCTURE OF A C PROGRAM

- **Documentation section** consists of a set of commentlines giving name of program, author and other details
- Single line comment is represented using //
- Multiple line comment is represented using `/*.....*/`
- Comment lines are not executable statements and therefore anything between `/*` and `*/` is ignored by compiler
- **Link section** provides instruction to the compiler to link functions from system library

STRUCTURE OF A C PROGRAM

- **Definition section** defines all symbolic constants
- There are some variables used in more than one function. Such variables are called global variables and are declared in **global declaration section** that is outside of all functions. This section also declares all user defined functions
- Every C program must have one **main() function** section
- This section contains two parts, **declaration part** and **executable part**

STRUCTURE OF A C PROGRAM

- C permits different forms of main statements. They are
 - `main ()`
 - `int main ()`
 - `void main ()`
 - `main (void)`
 - `void main (void)`
 - `int main (void)`
- Declaration part declares all variables used in executable part
- There is at least one statement in executable part

STRUCTURE OF A C PROGRAM

- These two parts must appear between opening and closing braces
- Program execution begins at opening brace and ends at closing brace
- Closing brace of main function section is logical end of program
- All statements in declaration and executable parts end with a semicolon(;)
- **Subprogram section** contains all user defined functions that are called in main function
- All sections, except main function section may be absent when they are not required

C Program to print Hello World

```
//This program prints Hello World
```

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

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

```
void main()
```

```
{
```

```
    clrscr();
```

```
    /*printf() function displays content that is  
passed between double quotes*/ printf("Hello  
World");
```

```
    getch();
```

```
}
```

C PROGRAM TOKENS

- Smallest individual unit in a program is known as tokens. C has following tokens
 - ▣ Keywords
 - ▣ Identifiers
 - ▣ Constants (literals)
 - ▣ Strings
 - ▣ Special Characters
 - ▣ Operators

Keywords

- These are words whose meaning has already been explained to C compiler
- Keywords cannot be used as variable names
- Also known as “**Reserved words**”
- There are only 32 keywords in C
- Examples:

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

Identifiers

- These are user-defined names
- C identifiers represent name in C program, for example, variables, functions, arrays, structures, unions, labels, etc
- An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore(_)
- It should not begin with any numerical digit
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive
- Commas or blank spaces cannot be specified within an identifier
- Keywords cannot be represented as an identifier
- Length of identifiers should not be more than 31 characters
- Identifiers should be written in such a way that it is meaningful, short, and easy to read

Identifiers Example

- **Example of valid identifiers**
 - Total
 - Sum
 - Average
 - `_m_`
 - `sum_1`
- **Example of invalid identifiers**
 - 2sum (starts with a numerical digit)
 - **int** (reserved word)
 - **char** (reserved word)
 - `m+n` (special character, i.e., '+')

Constants

- These are data items that never change their value during a program run. These are fixed values
- They are also called literals
- Types of Constants are
 - a) **Integer constant:** These are whole numbers without any fractional part. It must have at least one digit and must not contain any decimal point. It can be either positive or negative. Examples are 426, +200, -760
 - b) **Character constant:** A character constant is one character enclosed in single quotes. Examples are „A“, „5“, „=“. A character constant has corresponding ASCII values. For example ASCII value of „A“ is 65 and ASCII value of „a“ is 97

Constants

- c) **String constant:** Multiple character constants are treated as string constant
- ▣ A string constant is a sequence of characters surrounded by double quotes
 - ▣ Examples are “abcd”, ”seena”
 - ▣ Each string constant is by default (automatically) added with a special character “\0” which makes end of a string
 - ▣ Thus size of a string is Number of characters + null character (,\0”)
 - ▣ For example “abc” size is 4. Thus “abc” will be automatically represented as “abc\0” in memory. “\0” is an end-of-string marker

Constants

d) **Floating constant**: Floating constants are also called real constants. These numbers have fractional part. It may also have either positive or negative. Examples are 17.8, -13.867

Special Characters

- Following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose

`[] () { } , ; : * ... = #`

- **Brackets []:** These opening and closing brackets are used as array element reference
- **Braces {}:** Opening and closing curly braces are used to mark start and end of a block of code containing more than one statement
- **Comma (,):** To separate more than one statement
- **Semicolon (;):** Used at end of statements for termination

Special Characters

- **Parenthesis ()** : Are used to indicate function parameters & function calls
- **Asterick (*)**: This special symbol is used to create a pointer variable
- **Assignment Operator (=)**: For assigning values, this special symbol is used
- **Preprocessor (#)**: This you must have seen attached with the header files

Operators

- An operator is a symbol that tells computer to perform certain mathematical or logical manipulations
- Operators are used in programs to manipulate data and variables
- An expression is a sequence of operands and operators that reduces to a single value

Operators

- C operators are
 1. Arithmetic operators
 2. Relational operators
 3. Logical operators
 4. Assignment operators
 5. Increment and decrement operators
 6. Conditional operators
 7. Bitwise operators
 8. Special operators

ARITHMETIC OPERATORS

- C provides all basic arithmetic operators

<i>Operator</i>	<i>Meaning</i>
+	Addition or unary plus
−	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

- Integer division truncates any fractional part
- Modulo division operation produces remainder of an integer division
- Modulo division operator % cannot be used on floatingpoint data
- Example: $a - b$ $a + b$ $a * b$ a / b $a \% b$

Integer Arithmetic

- When both operands in a single arithmetic expression are integers, expression is called an integer expression, and operation is called integer arithmetic

- Integer arithmetic always yields an integer value

- If a and b are integers, then for $a = 14$ and $b = 4$
 $a - b = 10$

$$a + b = 18$$

$$a * b = 56$$

$$a / b = 3 \text{ (decimal part truncated)}$$

$$a \% b = 2 \text{ (remainder of division)}$$

Integer Arithmetic

- During modulo division sign of result is sign of first operand

$$-14 \% 3 = -2$$

$$-14 \% -3 = -2$$

$$14 \% -3 = 2$$

Real Arithmetic

- An arithmetic operation involving only real operands is called real arithmetic

- % cannot be used with real operands

$$x = 6.0/7.0 = 0.857143$$

$$y = 1.0/3.0 = 0.333333$$

$$z = -2.0/3.0 = -0.666667$$

Mixed mode arithmetic

- When one of operands is real and other is integer, expression is called a mixed-mode arithmetic expression
- If either operand is of real type, then only real operation is performed and result is always a real number

$$15/10.0 = 1.5$$

$$\text{Whereas } 15/10 = 1$$

RELATIONAL OPERATORS

- Comparisons can be done with the help of relational operators
- Value of a relational expression is either one or zero
- $10 < 20$ is true
- $20 < 10$ is false

<i>Operator</i>	<i>Meaning</i>
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

LOGICAL OPERATORS

- An expression which combines two or more relational expressions, is termed as a logical expression or a compound relational expression
- A logical expression also yields a value of one or zero

<i>op-1</i>	<i>op-2</i>	<i>Value of the expression</i>	
		<i>op-1 && op-2</i>	<i>op-1 op-2</i>
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

LOGICAL OPERATORS

Logical Operators		
Operator	Description	Example
&&	AND	x=6 y=3 x<10 && y>1 Return True
 	OR	x=6 y=3 x==5 y==5 Return False
!	NOT	x=6 y=3 !(x==y) Return True

ASSIGNMENT OPERATORS

- Assignment operators are used to assign result of an expression to a variable
- Usual assignment operator is „=“
- Example: `a=2`
It means assign value 2 to variable a
- Expression `a==5` is test or check whether a is equal to 5
- Expression `a=5` is assign 5 to variable a

ASSIGNMENT OPERATORS

- In addition, C has a set of „shorthand “ assignment operators of form

$v \text{ op} = \text{exp};$

is equivalent to $v = v \text{ op} (\text{exp});$

<i>Statement with simple assignment operator</i>	<i>Statement with shorthand operator</i>
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * (n+1)$	$a *= n+1$
$a = a / (n+1)$	$a /= n+1$
$a = a \% b$	$a \% = b$

INCREMENT(++) AND DECREMENT(--) OPERATORS

- Operator ++ adds 1 to operand, -- subtracts 1
- Both are unary operators
- Takes following form

Prefix

Postfix

++m;

m++;

--m;

m--;

- ++m; is equivalent to $m = m + 1$;
- --m; is equivalent to $m = m - 1$
- We use increment and decrement statements infor

and while loops extensively

INCREMENT(++) AND DECREMENT(--) OPERATORS

- `m = 5;`

- `y = ++m;`

- In this case, value of y and m would be 6

- Suppose, if we rewrite above statements as `m = 5;`

- `y = m++;`

- then, value of y would be 5 and m would be 6

- **Prefix operator** first adds result to operand and then result is assigned to variable on left

- **Postfix operator** first assign value to variable on left and

then increments operand

CONDITIONAL OPERATOR

- It is a **ternary operator** (3 operands)
- Conditional expression is of form
exp1 ? exp2 : exp3
where exp1, exp2, and exp3 are expressions
- exp1 is evaluated first
- If it is true, then exp2 is evaluated and become value of expression
- If exp1 is false, exp3 is evaluated and its value become value of expression

CONDITIONAL OPERATOR

- For example, consider following statements
a = 10;
b = 15;
x = (a > b) ? a : b;
- In this example, x will be assigned value of b ie, 15
- This can be achieved using the if..else statements
if (a > b)
x = a;
else
x = b;

BITWISE OPERATORS

- Bitwise operators are used for manipulation of data at bit level
- These operators are used for testing bits, or shifting them right or left

<i>Operator</i>	<i>Meaning</i>
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

BITWISE OPERATORS

<i>op1</i>	<i>op2</i>	<i>op1 & op2</i>	<i>op1 op2</i>	<i>op1 ^ op2</i>
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

■ Bitwise AND Example:

x - - -> 0000 0000 0000 1101

y - - -> 0000 0000 0001 1001

If we execute statement $z =$

$x \& y$;

then result would be:

z - - -> 0000 0000 0000 1001

BITWISE OPERATORS

□ Bitwise OR Example:

x - - -> 0000 0000 0000 1101

y - - -> 0000 0000 0001 1001

x |y -> 0000 0000 0001 1101

□ Bitwise Exclusive OR Example:

x - - -> 0000 0000 0000 1101

y - - -> 0000 0000 0001 1001

x ^y -> 0000 0000 0001 0100

BITWISE SHIFT OPERATORS

- Shift operators are used to move bit pattern either to left or to right
- Shift operators are represented by symbols << and >> and are used in following form:

Left shift: $op \ll n$

Right shift: $op \gg n$

- op is integer expression that is to be shifted and n is number of bit positions to be shifted

BITWISE SHIFT OPERATORS

■ $x = 0100\ 1001\ 1100\ 1011$

$x \ll 3 = 0100\ 1110\ 0101\ 1000$

$x \gg 3 = 0000\ 1001\ 0011\ 1001$

SPECIAL OPERATORS

- Comma operator
- sizeof operator
- Pointer operators (& and *)
- Member selection operators (. and →)

Comma operator

- Comma operator can be used to link related expressions together
- A comma-linked list of expressions are evaluated left to right and value of right-most expression is value of combined expression
- For example, statement `value = (x = 10, y = 5, x+y);`
It first assigns value 10 to x, then assigns 5 to y and finally assigns 15 to value

sizeof operator

- sizeof is a compile time operator and, when used with an operand, it returns number of bytes operand occupies
- Operand may be variable, constant or data type
- Example:

```
sum=23;
```

```
m=sizeof(sum)
```

It returns number of bytes variable sum occupies, ie, 4 bytes

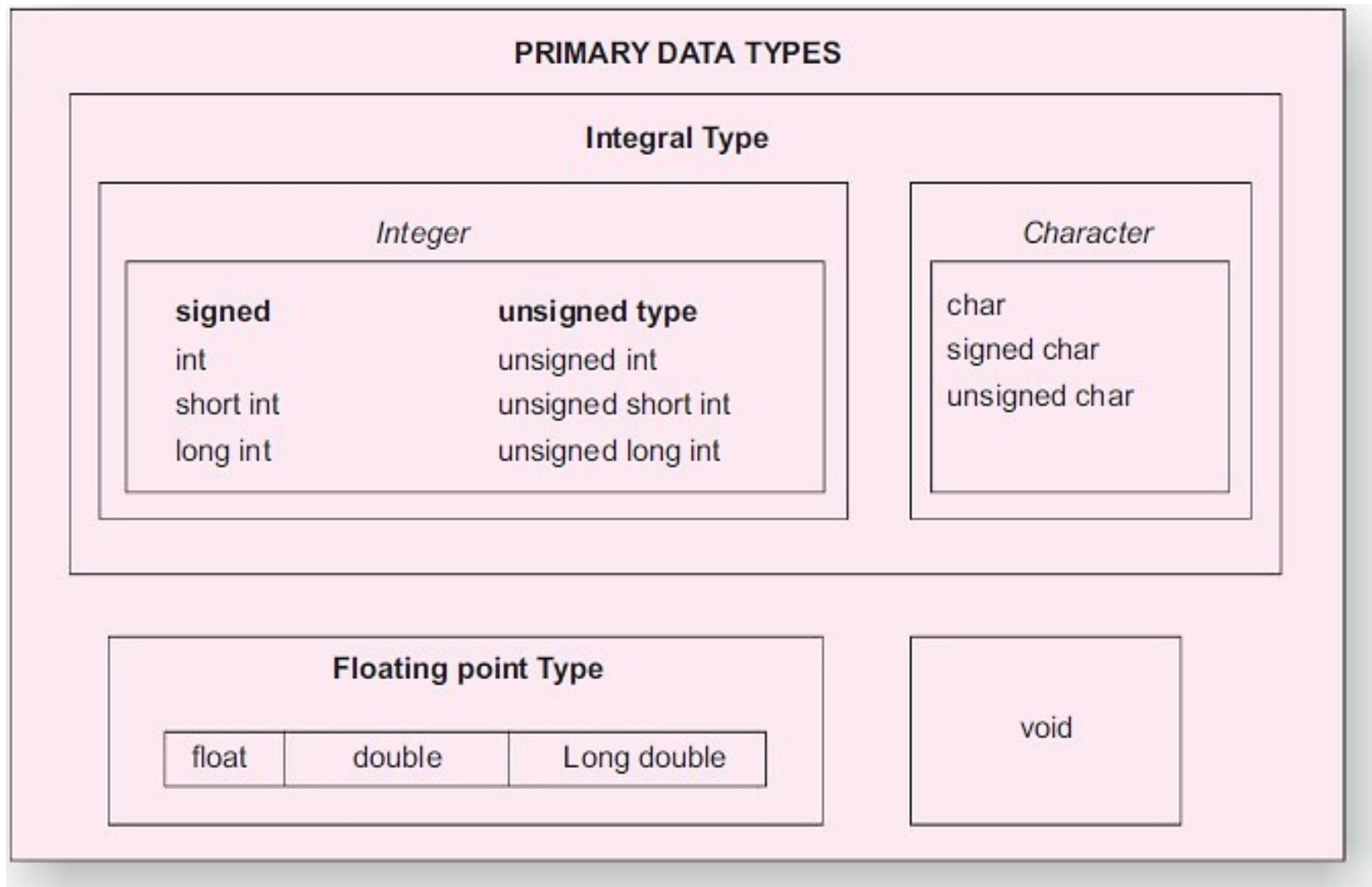
```
n=sizeof(int)
```

It assigns size of integer data type 2 to n

DATA TYPES

- Data type is a classification identifying one of various types of data. There are three classes of data types
 - ▢ Fundamental (primary) Data types
 - ▢ Derived Data types
 - ▢ User defined data types

Fundamental Data types



Fundamental data types

- int : for integers (2 byte memory space allocates in memory)
- char: for characters(1 byte memory space allocates in memory)
- float: for floating point numbers(4 byte memory space allocates in memory)
- double: for double precision floating point numbers(8 byte memory space allocates in memory)
- void: for empty set of values and non-returning functions. The void type has no value

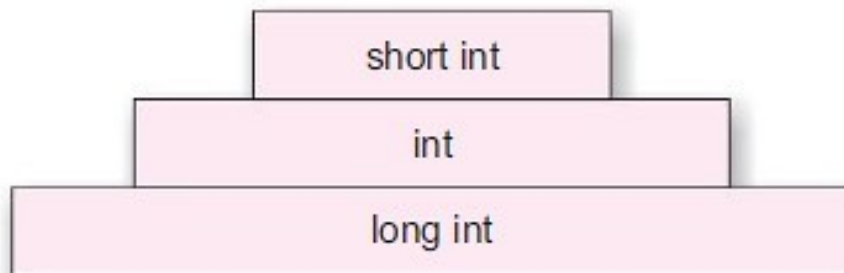
Fundamental data types

■ Size and Range of Basic Data Types

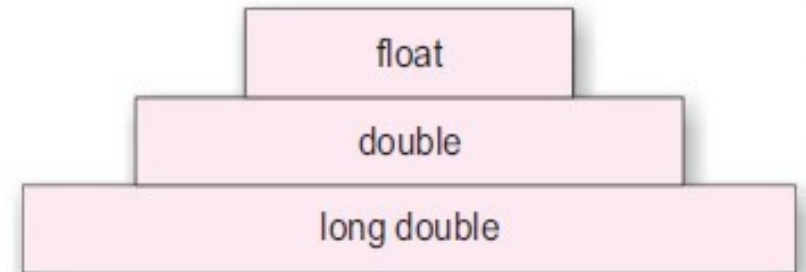
Type	Size (bits)	Range
char or signed char	8	−128 to 127
unsigned char	8	0 to 255
int or signed int	16	−32,768 to 32,767
unsigned int	16	0 to 65535
short int or		
signed short int	8	−128 to 127
unsigned short int	8	0 to 255
long int or		
signed long int	32	−2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	$3.4E - 38$ to $3.4E + 38$
double	64	$1.7E - 308$ to $1.7E + 308$
long double	80	$3.4E - 4932$ to $1.1E + 4932$

Fundamental data types

Integer types



Floating-point types



Derived data types

- Derived data types are constructed from fundamental data types. They are
 - **Arrays**: An array is a collection of values of same type that are referenced by a common name
 - **Functions**: A function is a named part of a program that can be invoked from other part of program
 - **Pointer**: A pointer is a variable that holds memory address. This address is usually location of another variable in memory

Derived data types

- ▣ **Constant**: keyword `const` can be added to declaration of an object to make that object a constant rather than a variable. Thus, variable of named constant cannot be altered during program run

Syntax

`const type name = value;`

Example: `const int a=10;`

User defined data types

- C allows programmers to define their identifier that would represent an existing data type. They are
 - ▣ typedef
 - ▣ structure
 - ▣ enumeration
 - ▣ union

typedef

- It allows users to define an identifier that would represent an existing data type.
- **Syntax**
typedef type identifier;
- Here, type refers to an existing data type and identifier refers to new name given to data type
- **Example:** typedef int units;
 typedef float marks;

Here, units symbolize int and marks symbolizes float.
They can belater used to declare variables as follows

typedef

```
units  batch1,  batch2;
```

```
marks name1,name2;
```

batch1 and batch2 are declared as int variables and name1 and name2 are declared as floating point variables

- Advantage of typedef is that we can create meaningful data type names for increasing readability of program

Enumeration

- Enumerated data type (also called enumeration or enum) provides a **way for attaching names to numbers** there by increasing comprehensibility of code. It has following form

enum identifier {value1,value2,.....,value n};

- identifier is a user defined enumerated data type which can be used to declare variables that can have one of values enclosed with in the braces(known as enumeration constant)

Enumeration

- Example:

- ❖ `enum week_day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};`

Here an enumerated data type `week_day` has been defined

- ❖ `week_day day1, day2;`

Here variables created of type `week_day`

- ❖ `day1=Wednesday;`

It is correct, `day1` can have any of above given 7 values

- ❖ `day2=7;`

It is incorrect, no other value can be assigned

Enumeration

- Enumerated means all the values are listed
- `enum` specifier automatically enumerates a list of words by assigning them values 0, 1, 2, 3, and so on
- Compiler automatically assigns integer digits beginning with 0 to all enumeration constants
- That is enumeration constant Monday is assigned 0, Tuesday is assigned 1 and so on

Enumeration

- We can give values explicitly (changing default original values) by following way

```
enum day {Monday=1, Tuesday, Wednesday=6, Thursday,  
Friday, Saturday, Sunday};
```

Now, Monday=1, Tuesday=2, Wednesday=6,
Thursday=7, Friday=8 etc.

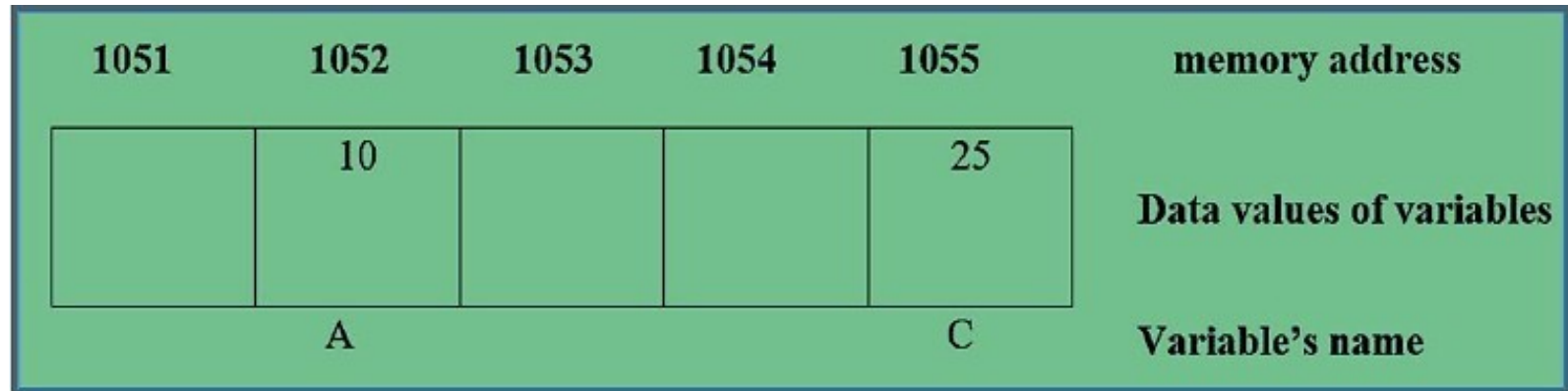
VARIABLES

- A variable is a **data name** that may be used to store a **data value**
- A variable may take different values at different times during execution
- Each variable has a specific storage location in memory where its value is stored
- Variables are called symbolic variables because these are named

VARIABLES

- There are two values associated with a symbolic variable
 - Data value: stored at some location in memory. This is sometimes referred to as a variable's r- value
 - Location value: This is address in memory at which its data value is stored. This is sometimes referred to as variable's l-value

VARIABLES



- r-value of A=10 and l-value of A=1052
- r-value of c=25 and l-value of c=1055
- Whenever we use assignment operator, expression to left of an assignment operator must be an l- value. That is, it must provide an accessible memory address where data can be written to

Variable Declaration

- A declaration associates a group of variables with a specific data type. Declaration of variables must be done before they are used in program

- Syntax

`Data_type variable_name;`

- Example: `int a;`

`float a, b, c;`

`char sex;`

Variable Initialization

- Process of giving initial values to variables is called initialization
- Example: `int x=10;`
`float n=22.889;`
`char answer='y';`

Expression

- An expression represents a single data item, such as a number or a character
- Expression can also represent logical conditions
- Example: $x+y$

$$y=z$$

$$x=y+z$$

- An expression statement consists of an expression followed by a semicolon
- For example, following two expression statements cause value of expression on right of equal sign to be assigned to variable on left

Expression

x=5;

$$x=y+z;$$

- A compound statement consists of several individual statements enclosed within a pair of braces ie; {and}

Algebraic Expression				C Expression
	$a \times b - c \times d$			$a * b - c * d$
	$(m + n) (a + b)$			$(m + n) * (a + b)$
	$3x^2 + 2x + 5$			$3 * x * x + 2 * x + 5$
	$\frac{a + b + c}{d + e}$			$(a + b + c) / (d + e)$

PRECEDENCE OF ARITHMETIC OPERATORS

- While executing an arithmetic statement, which has two or more operators, we may have some problems as to how exactly does it get executed
- For example, does the expression $2*x-3*y$ correspond to $(2x)-(3y)$ or to $2(x-3y)$?
- Similarly, does $A / B * C$ correspond to $A / (B * C)$ or to $(A / B) * C$?
- To answer these questions satisfactorily one has to understand the „hierarchy“ of operations

PRECEDENCE OF ARITHMETIC OPERATORS

- Priority or precedence in which operations in an arithmetic statement are performed is called hierarchy of operations
- Hierarchy of commonly used operators is shown below

Priority	Operators	Description
₁ st	* / %	multiplication, division, modular division
₂ nd	+ -	addition, subtraction
₃ rd	=	assignment

PRECEDENCE OF ARITHMETIC OPERATORS

- Example: Determine hierarchy of operations and evaluate following expression?

$$i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$$

- Stepwise evaluation of this expression is shown below:

i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8	operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8	operation: /
i = 1 + 1 + 8 - 2 + 5 / 8	operation: /
i = 1 + 1 + 8 - 2 + 0	operation: /
i = 2 + 8 - 2 + 0	operation: +
i = 10 - 2 + 0	operation: +
i = 8 + 0	operation: -
i = 8	operation: +

PRECEDENCE OF ARITHMETIC OPERATORS

- If there are more than one set of parentheses, operations within innermost parentheses would be performed first, followed by operations within secondinnermost pair and so on
- We must always remember to use pairs of parentheses
- Whenever parentheses are used, expressions within parentheses assume highest priority
- If two or more sets of parentheses appear one after another as shown above, expression contained in left most set is evaluated first and right most set in last

PRECEDENCE OF ARITHMETIC OPERATORS

- Evaluate expression9

$$-12/(3+3)*(2-1)$$

First pass

Step1: $9-12/6 * (2-1)$

Step2: $9-12/6 * 1$

Second pass

Step3: $9-2 * 1$

Step4: $9-2$

Third pass

Step5: 7

CONSOLE INPUT/OUTPUT OPERATIONS

1. `getchar()`

- `getchar()` function reads a single character from standard input
- It takes no parameters and its returned value is input character. It has following form

`variable name=getchar();`

- Example:

```
char c;
```

```
printf("Enter a character");
```

```
c=getchar();
```

CONSOLE INPUT/OUTPUT OPERATIONS

- Second line causes a single character to be entered from standard input device and then assigned to c
- Variable name has been declared as „char“ type

2. putchar()

- It displays a single character on screen
- This function takes one argument, which is character to be sent
- It also returns this character as its result

CONSOLE INPUT/OUTPUT OPERATIONS

- General form is

```
putchar(variable-name);
```

- Example:

```
char ans="y"  
putchar(ans);
```

3. gets()

- It receives a string from keyboard

CONSOLE INPUT/OUTPUT OPERATIONS

4. puts()

- Outputs a string to screen
- Example:

```
char vehicle [40];
```

```
Printf("Enter your vehicle name");
```

```
gets (vehicle);
```

```
puts (vehicle);
```

- These lines use gets and puts to transfer line of text into and out of computer

CONSOLE INPUT/OUTPUT OPERATIONS

5. printf()

- For outputting result we use printf() function
- Syntax

`printf ("formatted string", variable);`

- Example:

`a=3.5;`

`printf ("%f",a);`

`printf ("WELCOME MY FRIEND");`

CONSOLE INPUT/OUTPUT OPERATIONS

- Formatted strings are
 1. %d integers
 2. %f float
 3. %c character
 4. %s string
 7. %u unsigned integer
 9. %i signed decimal integer
 10. %p display a pointer
 11. %% prints a percent sign (%)

CONSOLE INPUT/OUTPUT OPERATIONS

6. scanf()

- It can read data from keyboard
- scanf () means “scan formatted”
- Syntax
scanf (“formatted string”, addressed variable);
- Example
scanf (“%d”, &num1);
scanf (“%d”, &num2);
scanf (“%d%d”, &num1, &num2);

MODULE 3

ARRAYS & STRINGS

ARRAYS

- An array is a **fixed size** sequenced collection of elements of **same data type**
- An array is a collection of variables of same datatype that are referenced by a **common name**
- **Array name** acts as a **pointer to zeroth element** of array
- TYPE OF ARRAYS
 - 1) One dimensional arrays
 - 2) Two- dimensional arrays
 - 3) Multi- dimensional arrays

ONE DIMENSIONAL ARRAY

- Like other variables an array needs to be declared so that compiler will know what kind of an array and how large an array we want

- General syntax is

type array_name [size];

- Eg: int marks [100];

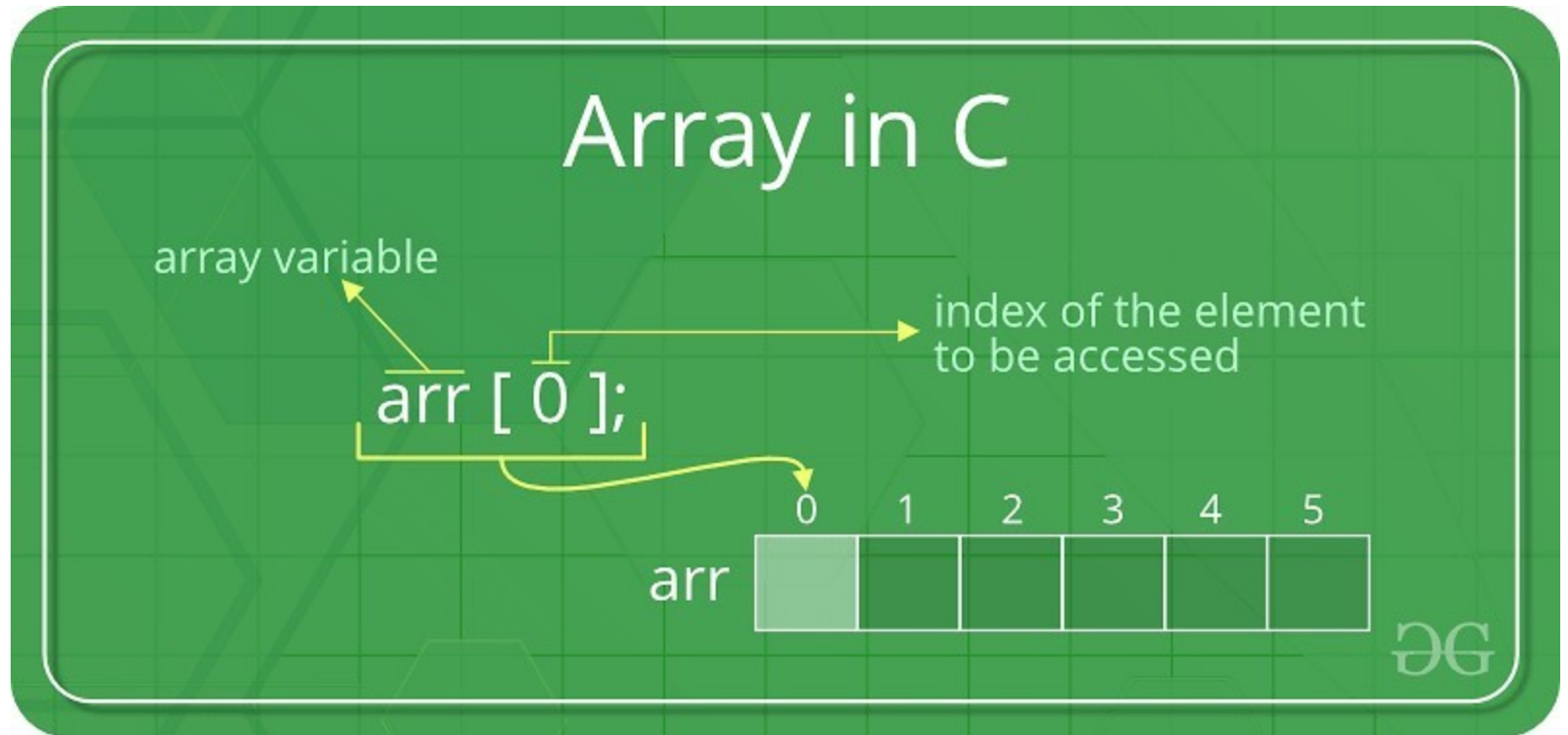
Computer reserves 100
contiguous storage locations as
shown

56	marks [0]
76	marks [1]
34	marks [2]
66	marks [98]
12	marks [99]

ONE DIMENSIONAL ARRAY

```
int arr[6];
```

Computer reserves 6 contiguous storage locations as shown



ONE DIMENSIONAL ARRAY

- An array is a collection of similar elements
- First element in the array is numbered 0, so last element is 1 less than size of array
- An array is also known as a **subscripted variable**
- Element numbers in [] are called **index** or **subscript**
- Before using an array its type and **dimension** must be declared
- However big an array its elements are always stored in contiguous memory locations

ONE DIMENSIONAL ARRAY

- C language treats character string, simply as array of characters
- Size in character string represents maximum number of characters that string can hold
- For example `char name[10];`
- Declares `name` as a character array (string) variable that can hold a maximum of 10 characters including null character

ONE DIMENSIONAL ARRAY

a

5	6	10	13	56	76	1	2	4	8
---	---	----	----	----	----	---	---	---	---



b

'a'	'b'	'c'	'd'	'e'
-----	-----	-----	-----	-----



c

'a'	'b'	1	5.6	'e'	34	2	3
-----	-----	---	-----	-----	----	---	---



Entering Data into an Array

- Here is section of code that places data into an array:
- ```
printf ("\n Enter 6 marks ") ;for
(i = 0 ; i <= 5 ; i++)
{
 scanf ("%d", &marks[i]) ;
}
```
- First time through loop, i has a value 0, so scanf ( ) function will cause value typed to be stored in the array element marks [0], first element of array. This process will be repeated until i become 49



# Printing Data from an Array

- Here is section of code that displays data from an array:
- ```
printf ( "\n Marks are" ) ; for  
( i = 0 ; i <= 5 ; i++ )  
{  
    printf ( "%d", marks[i] ) ;  
}
```
- First time through loop, i has a value 0, so printf () function will display value stored at array index 0 ie, element marks [0], first element of array. This process will be repeated until i become 49

INITIALIZATION OF 1D ARRAY

- An array can be initialized at either of following stages
 - At compile time
 - At run time

1) Compile time initialization:

Syntax type array_name[size]= {list of values};Eg:

```
int number [3] = {9, 5, 2};
```

```
float total [5] = {0.0, 15.75,-10.9};
```

INITIALIZATION OF 1D ARRAY

- Size may be omitted. In such cases, compiler allocates enough space for all initialized elements
 - Eg: `int counter [] = {1, 1, 1, 1};`
- Character arrays may be initialized in a similar manner
 - Eg: `char name [] = {„j“, „o“, „h“, „n“, „\0“};`
- If we have more initializers than declared size, compiler will produce an error
 - Eg: `int number [3] = {10, 20, 30, 40};` will not work. It is illegal in C

INITIALIZATION OF 1D ARRAY

2) Run time initialization:

We can use scanf () to initialize an array at runtime

Eg: int x [25],i;
 for (i=0;i<25;i++)
 scanf(“%d”,&x[i]);

LINEAR SEARCH

- Linear search is simplest search algorithm and often called **sequential search**
- In this type of searching, we simply traverse list completely and match each element of list with item whose location is to be found
- If match found then location of item is returned otherwise algorithm return NULL

LINEAR SEARCH EXAMPLE

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

LINEAR SEARCH EXAMPLE

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

LINEAR SEARCH EXAMPLE

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

BUBBLE SORT

- Sorting refers to ordering data in an increasing or decreasing fashion according to some linear relationship among data items
- Bubble sort is a simple **sorting algorithm**
- This sorting algorithm is comparison-based algorithm in which **each pair of adjacent elements is compared** and the elements are swapped if they are not in order
- This algorithm is **not suitable for large data sets** as its average and worst case complexity are of $O(n^2)$ where n is the number of items

BUBBLE SORT

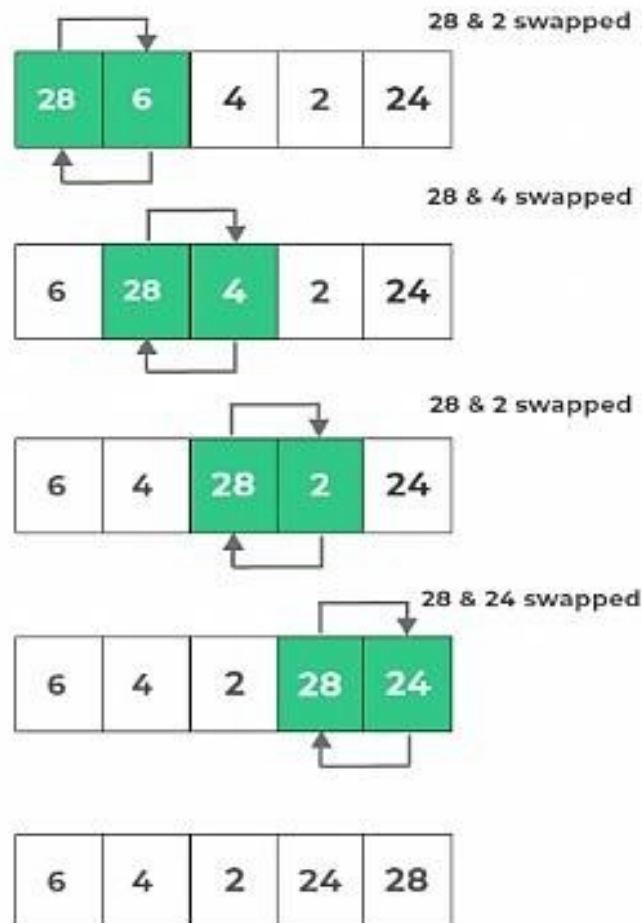
- If given array has to be sorted in **ascending order**, then bubble sort will start by comparing first element of array with second element
- If first element is greater than second element, it will **swap** both elements, and then move on to compare second and the third element, and so on
- If we have total **n elements**, then we need to repeat this process for **n-1 times**
- It is known as bubble sort, because with every complete iteration largest element in given array, bubbles up towards last place or highest index, just like a water bubble rises up to water surface

BUBBLE SORT EXAMPLE

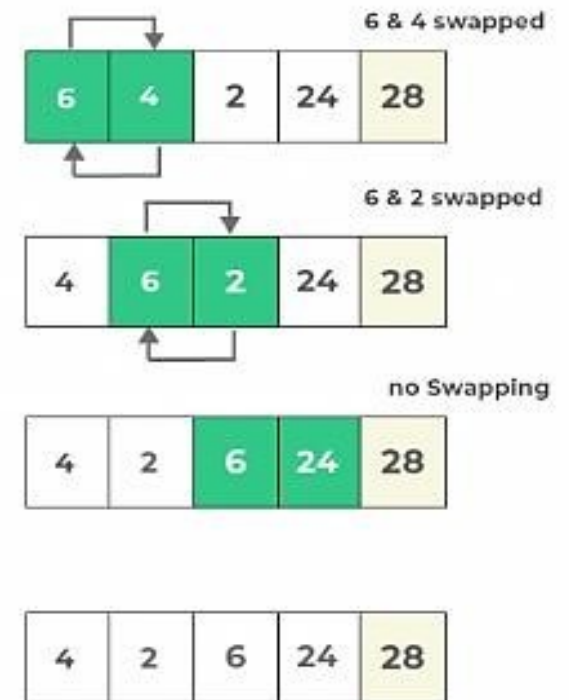
List

28	6	4	2	24
----	---	---	---	----

Pass 1

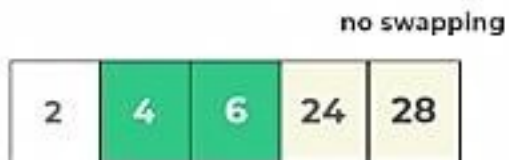
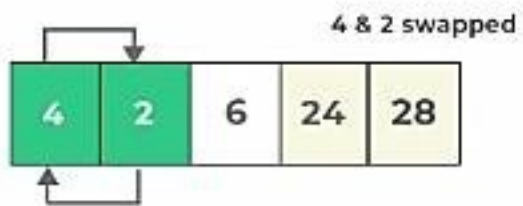


Pass 2

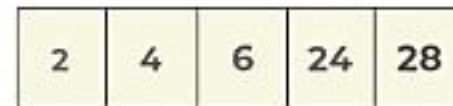
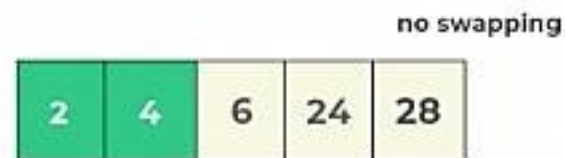


BUBBLE SORT EXAMPLE

Pass 3



Pass 4



Final Result



CHARACTER ARRAYS & STRINGS

- A string is a sequence of characters defined between double quotation marks

Eg: `printf (“WELL DONE”);`

Declaring and initializing

- Strings are declared as an array of characters
- Syntax is:

`char string_name[size];`

Eg: `char city [10];`

`char name[30];`

CHARACTER ARRAYS & STRINGS

- When compiler assigns a character string to a character array, it automatically supplies a null character (`\0`) at end of string
- Therefore size of a string
Size = maximum number of characters in string + one
- Character arrays can be initialized when they are declared

CHARACTER ARRAYS & STRINGS

- Initialization can be in either of following two forms

```
char city [9] ="NEW YORK";
```

```
char city[9]={„N“,„E“,„W“,„ „,„Y“,„O“,„R“,„K“,„\0“};
```

- Following format is also valid in C
- ```
char string[]= {„N“,„E“,„W“,„ „,„Y“,„O“,„R“,„K“,„\0“};
```

# READING STRING FROM KEY BOARD

## 1. Using scanf( ) function

```
char address[10]; scanf
("%s",address);
```

- In case of character arrays, ampersand (&) is not required before variable name
- Problem with scanf ( ) function is that it terminates its input on the first white space it finds
- If we typed NEW YORK then only string “NEW” will be read in to array address. Address array is created in memory as shown below

|   |   |   |    |   |   |   |   |   |   |
|---|---|---|----|---|---|---|---|---|---|
| N | E | W | \0 | ? | ? | ? | ? | ? | ? |
| 0 | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 | 9 |



# READING STRING FROM KEY BOARD

- Unused locations are filled with garbage value
- If we want the entire line “NEW YORK”, we use two character arrays of size  
`char ad1[5],ad2[5];`  
`scanf (“%s%s”,ad1,ad2);`
- • Then assign the string “NEW” to ad1 and “YORK” to ad2.
- `scanf ( )` is not capable of receiving multi-word strings

# READING STRING FROM KEY BOARD

- Way to avoid this limitation is by using function `gets ( )`
- It does not skip white space
- Usage of functions `gets( )` and its counterpart `puts( )` is shown below

```
void main ()
{
 char name[25] ;
 printf ("Enter full name\n ") ;
 gets (name) ;
 puts ("Hello!") ;
 puts (name) ;
}
```

## OUTPUT

```
Enter full name
Harun Shan
Hello!
Harun Shan
```

# STRING HANDLING FUNCTIONS

- String conversion functions are stored in header file `<string.h>`
  1. `strlen` - Finds length of a string
  2. `strlwr` - Converts a string to lowercase
  3. `strupr` - Converts a string to uppercase
  4. `strcat` - Appends one string at the end of another
  5. `strcpy` - Copies a string into another
  6. `strcmp` - Compares two strings
  7. `strdup` - Duplicates a string
  8. `strrev` - Reverses string

# strlen ( )

- This function counts number of characters present in a string

```
void main()
{
 char arr[] = "Newyear" ;
 int len1, len2 ;
 len1 = strlen (arr) ;
 len2 = strlen ("Humpty Dumpty") ;
 printf ("\nstring = %s length = %d", arr, len1) ;
 printf ("\nstring = %s length = %d", "Humpty Dumpty", len2) ;
}
```

## Output

string = Newyear length = 7

string = Humpty Dumpty length = 13

# strcpy( )

- This function copies contents of one string into another

```
void main()
{
 char source[] = "Soniya" ;
 char target[20] ;
 strcpy (target, source) ;
 printf ("\nsource string = %s", source) ;
 printf ("\ntarget string = %s", target) ;
}
```

## Output

source string = Soniya

target string = Soniya

# strcat( )

- This function concatenates source string at end of target string

```
void main()
{
 char source[] = "Brother" ;
 char target[30] = "Hello" ;
 strcat (target, source) ;
 printf ("\nsource string = %s", source) ;
 printf ("\ntarget string = %s", target) ;
}
```

## **Output**

```
source string = Brother target
string = HelloBrother
```

# strcmp( )

- This function compares two strings to find out whether they are same or different
- Two strings are compared character by character until there is a mismatch or end of one of strings is reached, whichever occurs first
- If two strings are identical, strcmp( ) returns a value zero
- If they're not, it returns numeric difference between ASCII values of first non-matching pair of characters

# strcmp( )

```
void main()
{
 char string1[] = "Jerry" ;
 char string2[] = "Ferry" ;
 int i, j, k ;
 i = strcmp (string1, "Jerry") ;
 j = strcmp (string1, string2) ;
 k = strcmp (string2, string1) ;
 printf ("\n%d %d %d", i, j, k) ;
}
```

## Output

0 4 -4

- In first call two strings are identical “Jerry” and “Jerry” and value returned by strcmp ( ) is zero
- In second call, result is 4, which is numeric difference between ASCII value of „J” and ASCII value of „F”
- In third call, result is -4, which is numeric difference between ASCII value of „F” and ASCII value of „J”



# <ctype.h>

- Character function use ctype.h header file. It is used for character testing and conversion functions
  - isalpha (c): Determine if argument is alphabetic. It return non zero value if true, 0 otherwise. Return type is int
  - isdigit (c): Determine if argument is a decimal digit. It return non zero value if true, 0 otherwise. Return type is int
  - islower (c): Determine if argument is lower case. It return non zero value if true, 0 otherwise. Return type is int

## <ctype.h>

- `isupper (c)`: Determine if argument is upper case. It return non zero value if true, 0 otherwise. Return type is `int`
- `tolower (c)`: Convert letter to lower case. Return type is `int`
- `toupper (c)`: Convert letter to upper case. Return type is `int`

# Example to print length of a string using strlen() library function and gets

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
 int i;
 char name[30];
 printf(" Enter some string");
 gets(name);
 i=strlen(name);
 printf("The length of string%d",i);
 getch();
}
```

# Example to print length of a string using gets( ) and null character

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,count=0;
 char name[30];
 printf(" Enter some string");
 gets(name);
 for(i=0;name[i]!='\0';i++)
 count++;
 printf("The length of string%d",count);
 getch();
}
```

# TWO DIMENSIONAL ARRAYS

- Two-dimensional array is also called a matrix
- Two dimensional arrays are declared as follows`type  
array_name [row-size] [column-size];`

- Example:

```
int stud[4][2] = {
```

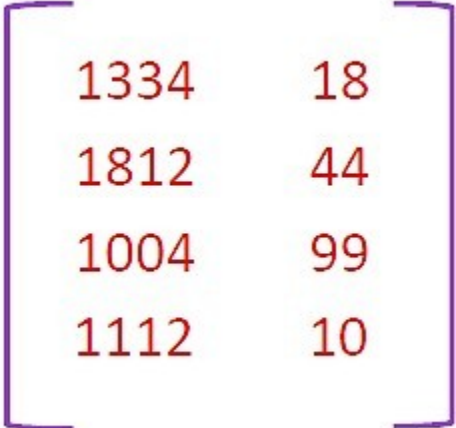
```
 { 1334, 18 },
```

```
 { 1812, 44 },
```

```
 { 1004, 99 },
```

```
 { 1112, 10 }
```

```
};
```



|      |    |
|------|----|
| 1334 | 18 |
| 1812 | 44 |
| 1004 | 99 |
| 1112 | 10 |

OR

```
int stud[4][2] = { 1334, 18, 1812, 44, 1004, 99, 1112, 10 } ;
```

# TWO DIMENSIONAL ARRAYS

- It is important to remember that while initializing a 2-D array it is necessary to mention second (column) dimension, whereas first dimension (row) is optional
- Thus declarations,  
`int arr[2][3] = { 52, 30, 23, 55, 56, 85 } ;`  
`int arr[ ][3] = { 52, 30, 23, 55, 56, 85 } ;`  
are perfectly acceptable
- Whereas,  
`int arr[2][ ] = { 52, 30, 23, 55, 56, 85 } ;`  
`int arr[ ][ ] = { 52, 30, 23, 55, 56, 85 } ;`  
would never work

# Memory Map of a 2-Dimensional Array

- Array elements have been stored row wise and accessed row wise
- We can access array elements column wise as well
- Traditionally, array elements are being stored and accessed row wise

| <b>s[0][0]</b> | <b>s[0][1]</b> | <b>s[1][0]</b> | <b>s[1][1]</b> | <b>s[2][0]</b> | <b>s[2][1]</b> | <b>s[3][0]</b> | <b>s[3][1]</b> |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| <b>1334</b>    | <b>18</b>      | <b>1812</b>    | <b>44</b>      | <b>1004</b>    | <b>99</b>      | <b>1112</b>    | <b>10</b>      |
| <b>65508</b>   | <b>65510</b>   | <b>65512</b>   | <b>65514</b>   | <b>65516</b>   | <b>65518</b>   | <b>65520</b>   | <b>65522</b>   |

# Memory Map of a 2-Dimensional Array

- Array arrangement shown below is only conceptually true
- This is because memory doesn't contain rows and columns
- In memory whether it is a one-dimensional or a two-dimensional array, elements are stored in one continuous chain
- Arrangement of array elements of a two-dimensional array in memory is shown in figure



**END.....**

# MODULE 4

## WORKING WITH FUNCTIONS

# FUNCTION

- A function is a named part of a program that can be invoked from other part of program
- That is, a function is a self-contained block of statement that performs a coherent task of some kind
- Every C program can be thought of as a collection of these functions
- A function can be classified in to two categories

# FUNCTION

**1. Library functions:** These are pre-defined. Library functions are not required to be written by us

- Examples are printf () ,scanf (), sqrt (), cos (), strcat () etc.
- This library of functions is present on disk and is written for us by people who write compilers for us
- Almost always a compiler comes with a library of standard functions
- Procedure of calling both types of functions is exactly same

# FUNCTION

**2. User defined functions:** This type of functions has to be developed by user at time of writing a program

- `main ()` is an example of user defined function

# FUNCTION

- A function has three parts
  1. Function declaration/prototype  
Syntax: `type function_name (argument list);`
  2. Function call  
Syntax: `function_name(argument list);`
  3. Function definition

# FUNCTION

## ❖ Function definition

### Syntax:

```
type function-name (parameter/argument list)
{
 Local variable declarations;
 executable statement 1;
 executable statement 2;

 return statement;
}
```

# FUNCTION HEADER

- type function-name (parameter list) is called as function header
- Statement with in opening and closing braces are function body
- Semicolon is not used at end of function header
- A function must be declared before main () function
- Function calling is done with in main () function
- Parameter is also called as arguments



# A SIMPLE FUNCTION

```
#include<stdio.h>
#include<conio.h>
void message(); /* function declaration*/
void main()
{
 message() ; /* function call*/
 printf ("Hai...It's Monday!") ;
 getch();
}

void message() /* function definition*/
{
 printf ("Hai...It's Tuesday") ;
}
```

## OUTPUT

Hai...It's Tuesday  
Hai...It's Monday!

# A SIMPLE FUNCTION

- Here, main() itself is a function and through it we are calling function message()
- main() becomes **calling function**, whereas message() becomes **called function**
- Any C program contains at least one function
- If a program contains only one function, it must be main ()
- If a C program contains more than one function, then one (and only one) of these functions must be main (), because program execution always begins with main ()

# A SIMPLE FUNCTION

- After each function has done its thing, control returns to `main ( )`
- When `main ( )` runs out of function calls, program ends
- C program is a collection of one or more functions
- A function gets called when function name is followed by a semicolon
- A function can call itself. Such a process is called **‘recursion’**
- A function can be called from other function, but a **function cannot be defined in another function**

# Small Program Using return Statement

```
#include<stdio.h>
#include<conio.h>
int mul(int, int); /* function declaration*/
void main ()
{
 int y,a,b;
 printf("Enter two integer values");
 scanf("%d%d",&a,&b);
 y=mul (a, b); /* function call using actual parameters a and b*/
 printf ("%d\n",y);
 getch();
}

int mul(int x, int y) /* function definition using formal parameters(dummy variables) */
{
 int p; /* local variable declaration*/
 p=x*y;
 return(p);
}
```

# Formal parameter v/s Actual parameter

- **Formal Parameter** : A variable appear in the prototype of the function or method
- **Actual Parameter** : Variable or expression corresponding to a formal parameter that appears in function or method call in calling environment
- In above example: a and b are actual parameters  
x and y are formal parameters

# return Statement

- return statement serves two purposes:
  1. On executing return statement it immediately transfers control back to calling program
  2. It returns value present in parentheses after return, to calling program
- In above program value of sum of three numbers is being returned
- All the following are valid return statements

```
return (a) ; return
(23) ; return (
16.94) ; return ;
```

# return Statement

- If we want that a called function should not return any value, in that case, we must mention so by using **keyword void** as shown below

```
void display()
{
 printf ("Hello every one") ;
}
```

- In absence of return statement, closing brace act as a void return

# Declarations

- `int a,b;` - Variable declaration
- `int a[20];` - Array declaration (One Dimensional)
- `int a[4][4];` - Array declaration (Two Dimensional)
- `char a[25];` - String declaration

## Function Declarations

- `void add(int,int);`
- `int add (int,float);`
- `float add(float,float);`
- `void add ();`



# Example for Function without argument and return value

```
#include<stdio.h>
#include<conio.h>
void sum();
void main()
{
 printf("\nGoing to calculate the sum of two numbers:");
 sum();
 getch();
}
void sum()
{
 int a,b;
 printf("\nEnter two numbers");
 scanf("%d %d",&a,&b);
 printf("The sum is %d",a+b);
}
```

## OUTPUT

```
Going to calculate the sum of two numbers:

Enter two numbers

10 24

The sum is 34
```

# Example for Function without argument and with return value

```
#include<stdio.h>
#include<conio.h>
int square();
void main()
{
 float area;
 printf("Going to calculate the area of the square\n");
 area = square();
 printf("The area of the square: %f\n",area);
 getch();
}
int square()
{
 float side;
 printf("Enter the length of the side in meters: ");
 scanf("%f",&side);
 return side * side;
}
```

## OUTPUT

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

# Example for Function with argument and without return value

```
#include<stdio.h>
#include<conio.h>
void average(int, int, int, int, int);
void main()
{
 int a,b,c,d,e;
 printf("\nGoing to calculate the average of five numbers:");
 printf("\nEnter five numbers:");
 scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
 average(a,b,c,d,e);
 getch();
}
void average(int a, int b, int c, int d, int e)
{
 float avg;
 avg = (a+b+c+d+e)/5;
 printf("The average of given five numbers : %f",avg);
}
```

## OUTPUT

```
Going to calculate the average of five numbers:

Enter five numbers: 10 20 30 40 50

The average of given five numbers: 30.000000
```

# Example for Function with argument and with return value

```
#include<stdio.h>
#include<conio.h>
int sum(int, int);
void main()
{
 int a,b,result;
 printf("\nGoing to calculate the sum of two numbers:");
 printf("\nEnter two numbers:");
 scanf("%d %d",&a,&b);
 result = sum(a,b);
 printf("\nThe sum is : %d",result);
 getch();
}
int sum(int a, int b)
{
 return a+b;
}
```

## OUTPUT

```
Going to calculate the sum of two numbers:
Enter two numbers: 10 20
The sum is : 30
```

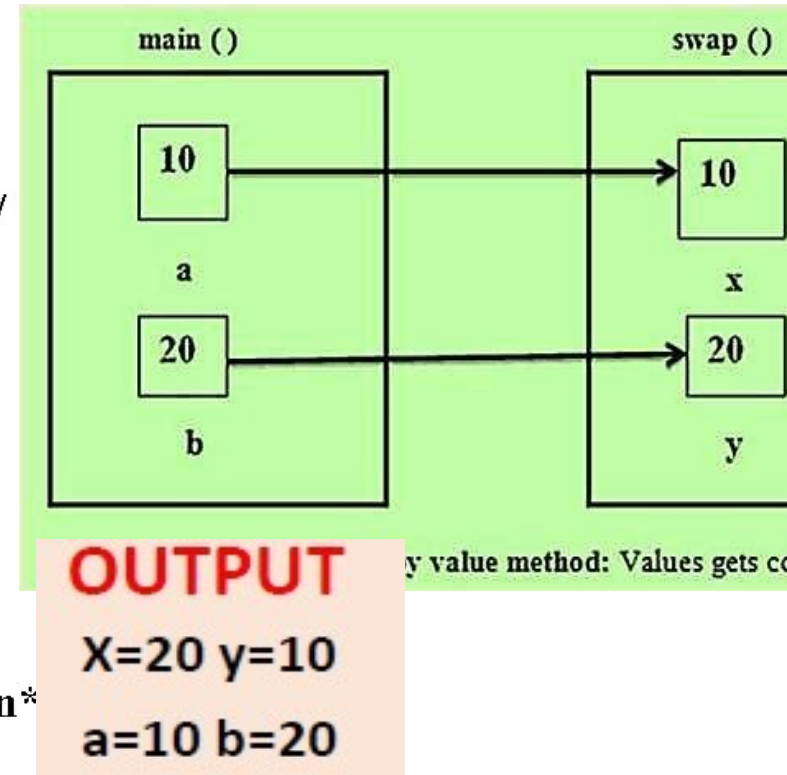
# PASS BY VALUE ( CALL BY VALUE )

- In this method 'value' of each of actual arguments (arguments in function call statement) in calling function is copied into corresponding formal arguments (arguments in function definition section) of called function
- Function creates its own copy of argument values and then uses them
- With this method changes made to the formal arguments in called function have no effect on values of actual arguments in calling function
- That is, any change in formal parameter is not reflected back to actual parameters

# PASS BY VALUE ( CALL BY VALUE )

```
#include<stdio.h>
#include<conio.h>
void swap(int, int); /* function declaration*/
void main()
{
 int a = 10, b = 20 ;
 swap (a, b) ; /* function call*/
 printf ("\na = %d b = %d", a, b) ;
 getch();
}

void swap (int x, int y) /* function definition*
{
 int temp ;
 temp = x ;
 x = y;
 y = temp;
 printf ("\nx = %d y = %d", x, y) ;
}
```



Note that values of a and b remain unchanged even after exchanging values of x and y



# PASSING SINGLE 1D ARRAYELEMENT TO A FUNCTION

```
#include <stdio.h>
#include <conio.h>
void display(int);
void main()
{
 int i ;
 int marks[] = { 55, 65, 75, 56, 78, 78, 90 } ;
 for (i = 0 ; i <= 6 ; i++)
 display (marks[i]) ;
 getch();
}

void display (int m)
{
 printf ("%d ", m) ;
}
```

## OUTPUT

55 65 75 56 78 78 90

since at a time only one element is being  
passed, this element is  
collected in an ordinary integer  
variable m, in function display( )

# PASSING COMPLETE 1D ARRAY TO A FUNCTION

```
#include<stdio.h>
float findAverage(int marks[])
{
 int i, sum = 0;
 float avg;
 for (i = 0; i <= 4; i++) {
 sum += marks[i];
 }
 avg = (sum / 5);
 return avg;
}
void main()
{
 float avg;
 int marks[] = {99, 90, 96, 93, 95};
 avg = findAverage(marks); // name of the array is passed as argument
 printf("Average marks = %f", avg);
}
```



# PASSING A MULTI-DIMENSIONAL ARRAY TO A

```
#include <stdio.h>
#include <conio.h>
void show(int q[][4], int row, int col);
void main()
{
 int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 6} ;
 show (a, 3, 4) ;
 getch();
}

void show (int q[][4], int row, int col)
{
 int i, j ;
 for (i = 0 ; i < row ; i++)
 {
 for (j = 0 ; j < col ; j++)
 {
 printf ("%d\t", q[i][j]) ;
 }
 printf ("\n") ;
 }
}
```

## OUTPUT

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

# FUNCTION

# RECURSION

- A function is called 'recursive' if a statement within body of a function calls same function
- A function can call itself. Such a process is called 'recursion'

## EXAMPLE

- Factorial of a number is product of all integers between 1 and that number
- For example, 4 factorial is  $4 * 3 * 2 * 1$
- This can also be expressed as  $4! = 4 * 3!$  Where '!' stands for factorial
- Thus factorial of a number can be expressed in form of itself
- Hence this can be programmed using recursion

# RECURSION

```
#include <stdio.h>
#include <conio.h>
int fact (int);
void main()
{
 int n,f;
 printf("Enter the number whose factorial you want to calculate?");
 scanf("%d",&n);
 f = fact(n);
 printf("factorial = %d",f);
 getch();
}
int fact(int n)
{
 if (n==0)
 {
 return 1;
 }
 else if (n == 1)
 {
 return 1;
 }
 else
 {
 return n*fact(n-1);
 }
}
```

## OUTPUT

Enter the number whose factorial you want to calculate?5

Factorial = 120

**Function call: fact(5)**

```
return 5 * fact(4) = 120
├── return 4 * fact(3) = 24
│ ├── return 3 * fact(2) = 6
│ │ ├── return 2 * fact(1) = 2
│ │ └── return 1 = 1
└── 5! = 1*2*3*4*5 = 120
```

# STORAGE CLASSES

- Storage classes provide information about variable's location and visibility
- Variable's storage class tells us
  - Where variable would be stored?
  - What will be initial value of variable, if initial value is not specifically assigned?
  - What is scope of variable; i.e. in which function value of variable would be available?
  - What is life of variable; i.e. how long would the variable exist?

# STORAGE CLASSES

- **Scope** of variable determines over what region of program a variable is actually available for use
- **Visibility** refers to accessibility of a variable from memory
- **Longevity** refers to period during which a variable retains a given value during execution of a program
- **Life time** of a variable is duration of time in which a variable exists in memory during execution

# STORAGE CLASSES

- There are four storage classes in C:
  1. Automatic
  2. Register
  3. Static
  4. External

# Automatic Storage Class

- A variable declared inside a function by default, automatic
- They are created when function is called and destroyed automatically when function is exited
- We can write explicitly as **auto int number;**

**Storage**

– Memory.

**Default initial value**

– An unpredictable value, which is often called a garbage value.

**Scope**

– Local to the block in which the variable is defined.

**Life**

– Till the control remains within the block in which the variable is defined.



# Static Storage Class

- Static variable initialized once when program is compiled
- Value of static variable cannot change

**static int x;**

|                       |                                                                    |
|-----------------------|--------------------------------------------------------------------|
| Storage               | – Memory                                                           |
| Default initial value | – Zero                                                             |
| Scope                 | – Local to the block in which the variable is defined.             |
| Life                  | – Value of the variable persists between different function calls. |

# AUTO & STATIC – A COMPARISON

```
main()
{
 increment() ;
 increment() ;
 increment() ;
}

increment()
{
 auto int i = 1 ;
 printf ("%d\n", i) ;
 i = i + 1 ;
}
```

```
main()
{
 increment() ;
 increment() ;
 increment() ;
}

increment()
{
 static int i = 1 ;
 printf ("%d\n", i) ;
 i = i + 1 ;
}
```

The output of the above programs would be:

1  
1  
1

1  
2  
3

# External Storage Class

- External variable's scope is global, not local
- External variables are declared outside all functions, yet are available to all functions that care to use them
- We can write explicitly as **extern int y;**

|                              |                                                                 |
|------------------------------|-----------------------------------------------------------------|
| <b>Storage</b>               | – Memory.                                                       |
| <b>Default initial value</b> | – Zero.                                                         |
| <b>Scope</b>                 | – Global.                                                       |
| <b>Life</b>                  | – As long as the program's execution<br>doesn't come to an end. |

# External Storage Class

```
int fun1(void);
int fun2(void);
int fun3(void);
int x ; /* global */
main()
{
 x = 10 ; /* global x */
 printf("x = %d\n", x);
 printf("x = %d\n", fun1());
 printf("x = %d\n", fun2());
 printf("x = %d\n", fun3());
}
int fun1(void)
{
 x = x + 10 ;
}
```

```
int fun2(void)
{
 int x ; /* local */
 x = 1 ;
 return (x);
}
int fun3(void)
{
 x = x + 10 ; /* global x */
}
```

## Output

```
x = 10
x = 20
x = 1
x = 30
```

# External Storage Class

```
main()
{
 extern int y; /* external declaration */

}
func1()
{
 extern int y; /* external declaration */

}
```

# Register Storage Class

- We can tell compiler that a variable should be kept in one of machine's registers, instead of keeping in memory
- Register access is faster than memory access

**register int count;**

|                              |                                                                               |
|------------------------------|-------------------------------------------------------------------------------|
| <b>Storage</b>               | - CPU registers.                                                              |
| <b>Default initial value</b> | - Garbage value.                                                              |
| <b>Scope</b>                 | - Local to the block in which the variable is defined.                        |
| <b>Life</b>                  | - Till the control remains within the block in which the variable is defined. |

# STRUCTURE

- A structure is a collection of variables of different data types that are referenced by a common name
- A structure contains a number of data types grouped together
- These data types may or may not be of same type

## Structure declaration Syntax

```
struct <structure name>
{
 structure element 1 ;
 structure element 2 ;
 structure element 3 ;

};
```

# STRUCTURE

- Once new structure data type has been defined one or more variables can be declared to be of that type

```
struct book
{
 char name ;
 float price ;
 int pages ;
} ;
struct book b1, b2, b3 ;
```

**is same as**

```
struct book
{
 char name ;
 float price ;
 int pages ;
} b1, b2, b3 ;
```



# Structure Initialization

- Like variables and arrays, structure variables can also be initialized where they are declared
- Closing brace in structure type declaration must be followed by a semicolon

```
struct book
{
 char name[10] ;
 float price ;
 int pages ;
};
struct book b1 = { "Chemistry", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

# ACCESSING STRUCTURE ELEMENTS

- In arrays we can access individual elements of an array using a subscript. Structures use a different scheme
- They use a **dot (.) operator**
- So to refer to pages of structure defined in sample program we have to use,  
**b1.pages**
- Similarly, to refer to price we would use,  
**b1.price**
- Note that before dot there must always be a structure variable and after dot there must always be a structure element

# ASSIGNING VALUES TO STRUCTUREELEMENTS

```
struct book-bank
{
char title[20];
char author[15];
int pages ;
float price;
} book1, book2, book3;
```

We can assign values to the members of book1.

```
strcpy (book1.title,"C++");
strcpy (book1.author,"XYZA");
book1.pages=290;
book1.price=320.50;
```

We can also use scanf to give the values through the keyword.

```
scanf ("%s\n",book1.title);
scanf ("%d\n",&book1.pages);
```

# ARRAY OF STRUCTURES

- Array of structures: an array of structure data types which themselves are a collection of dissimilar data types

```
struct student
{
 int x;
 float m;
} a [3] = {{10,3.5}, {5, 1.2}, {25, 0.07}};
```

# ARRAY OF STRUCTURES

```
#include<stdio.h>
#include<conio.h>
void main()
{
 struct book
 {
 char name ;
 float price ;
 int pages ;
 } ;
 struct book b[200] ;
 int i ;
 for (i = 0 ; i <= 199 ; i++)
 {
 printf ("\nEnter name, price and pages ") ;
 scanf ("%c %f %d", &b[i].name, &b[i].price, &b[i].pages) ;
 }
 for (i = 0 ; i <= 199 ; i++)
 printf ("\n%c %f %d", b[i].name, b[i].price, b[i].pages) ;
 getch();
}
```

# STRUCTURE ASSIGNMENT

- Values of a structure variable can be assigned to another structure variable of same type using assignment operator
- It is not necessary to copy structure elements piece-meal
- Obviously, programmers prefer assignment to piece-meal copying
- This is shown in following example

# STRUCTURE ASSIGNMENT

```
#include<stdio.h>
#include<conio.h>
void main()
{
 struct employee
 {
 char name[20] ;
 int age ; float salary ;
 } ;
 struct employee e1 = { "Sanjay", 30, 5500.50 } ;
 struct employee e2, e3 ;
 /* piece -meal copying */
 strcpy (e2.name, e1.name) ;
 e2.age = e1.age;
 e2.salary = e1.salary;
 /* copying all elements at one go */
 e3 = e2;
 printf ("\n%s %d %f", e1.name, e1.age, e1.salary) ;
 printf ("\n%s %d %f", e2.name, e2.age, e2.salary) ;
 printf ("\n%s %d %f", e3.name, e3.age, e3.salary) ;
 getch();
}
```

## OUTPUT

```
Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000
```

# PASSING STRUCTURE TO A FUNCTION

```
#include<stdio.h>
#include<conio.h>
struct book
{
 char name[25] ;
 char author[25] ;
 int callno ;
} ;
void main()
{
 struct book b1 = { "Programming in C ", "James", 9952489217 } ;
 display (b1) ;
 getch();
}
display (struct book b)
{
 printf ("\n%s %s %d", b.name, b.author, b.callno) ;
}
```

## OUTPUT

Programming in C James 9952489217



# UNION

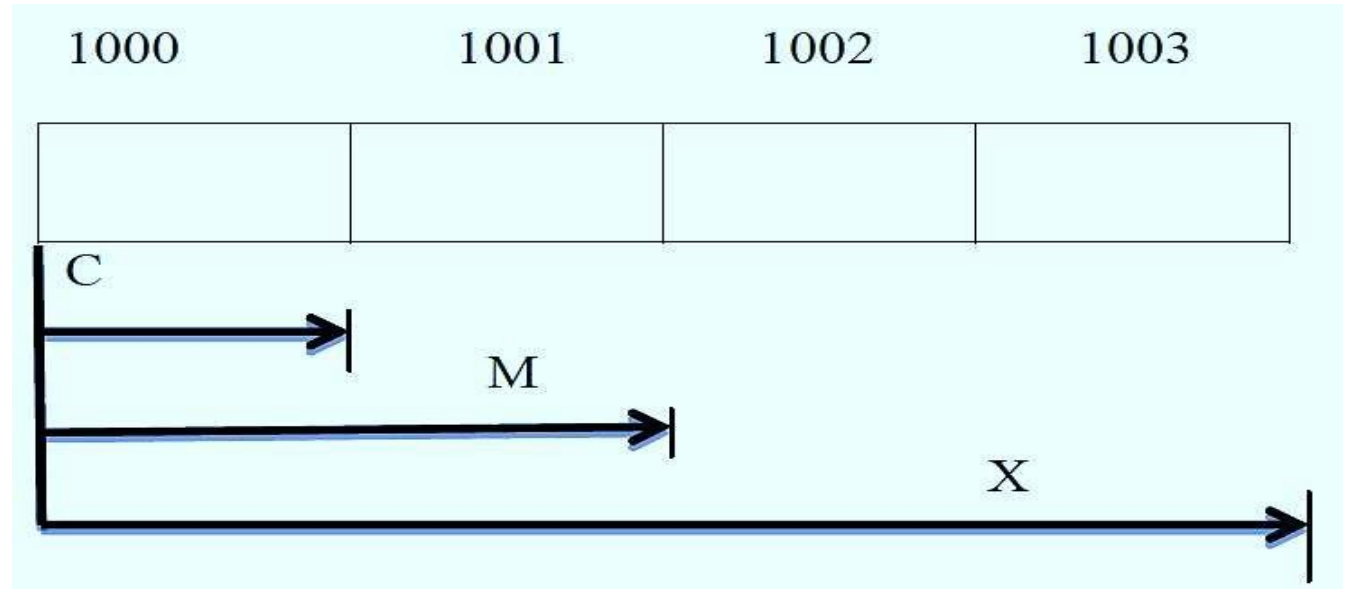
- Syntax is same as structures
- In structure each member has its own storage location, whereas **all members of a union use same location**
- It can **handle only one member at a time**
- Union may be used in all places where a structure is allowed
- To access a union member, we can use same syntax that we use for structure members
- That is,  
code.m code.x code.c all are valid

# UNION

- Compiler allocates a piece of storage that is large enough to hold largest variable type in union
- In below declaration, member x requires 4 byte memory
- So only 4 byte memory is allocated

**union item**

```
{
 int m;
 float x;
 char c;
} code;
```



**END...**

# MODULE 5

## POINTERS & FILES

# POINTER

- A pointer is a variable that holds memory address of location of another variable in memory
- It is a **derived data type** in C
- Consider declaration,

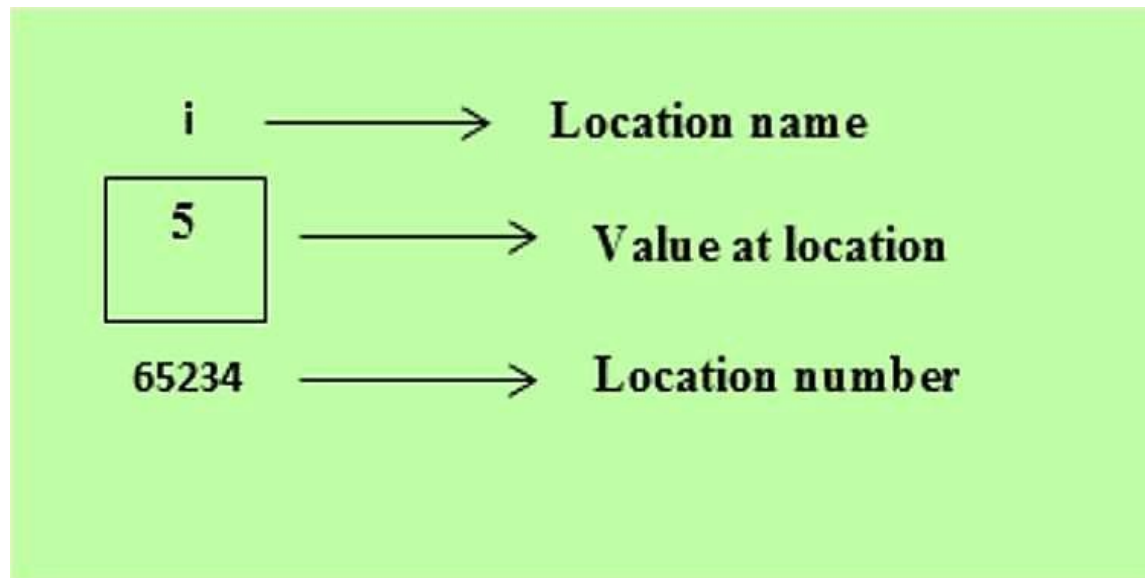
**int i = 5 ;**

This declaration tells C compiler to:

- **Reserve space in memory to hold integer value**
- **Associate name i with this memory location**
- **Store value 5 at this location**

# POINTER

- We may represent **i**'s location in memory by following memory map



- We see that computer has selected memory location **65234** as place to store value **5**

# POINTER

- Important point is, i's address in memory is a number. We can print this address number through following program:

```
#include<stdio.h>
#include<conio.h>
void main ()
{
 int i = 5 ;
 Printf ("\nAddress of i = %u", &i);
 Printf ("\nValue of i = %d", i);
}
```

## OUTPUT

**Address of i = 65234**

**Value of i = 5**

- **&** used in this statement is C's address of operator. Expression **&i** returns address of variable **i**
- **%u** is a format specifier for printing an **unsigned integer**

# POINTER

- Other pointer operator available in C is ‘\*’, called ‘**value at address operator**’. It gives value stored at a particular address
- ‘Value at address’ operator is also called ‘**indirection**’ operator or ‘**pointer operator**’

```
#include<stdio.h>
#include<conio.h>
void main ()
{
 int i = 3 ;
 printf ("\nAddress of i = %u", &i) ;
 printf ("\nValue of i = %d", i) ;
 printf ("\nValue of i = %d", *(&i)) ;
 getch();
}
```

## OUTPUT

**Address of i = 65234**

**Value of i = 3**

**Value of i = 3**



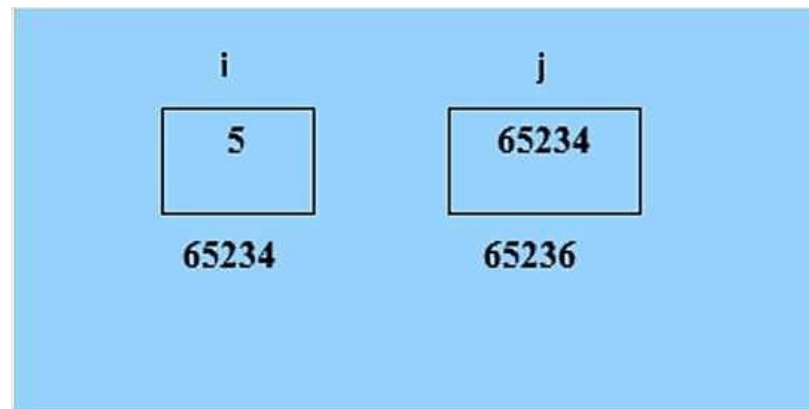
# POINTER

- Expression **&i** gives address of variable i. This address can be collected in a variable, by saying,

**j = &i;**

- But remember that j is not an ordinary variable like any other integer variable. It is a variable that contains address of other variable (i in this case). Since j is a variable compiler must provide it space in memory

**i's value is 5 and j's value is i's address**



# POINTER

- we can't use `j` in a program without declaring it. And since `j` is a variable that contains address of `i`, it is declared as, `int *j` ;
- This declaration tells compiler that `j` will be used to store address of an integer value. In other words `j` points to an integer
- Let us go by meaning of `*`. It stands for 'value at address'
- Thus, `int *j` would mean, value at address contained in `j` is an `int`

# POINTER EXAMPLE

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i = 5 ;
 int *j ;
 j = &i ;
 printf ("\nAddress of i = %u", &i) ;
 printf ("\nAddress of i = %u", j) ;
 printf ("\nAddress of j = %u", &j) ;
 printf ("\nValue of j = %u", j) ;
 printf ("\nValue of i = %d", i) ;
 printf ("\nValue of i = %d", *(&i)) ;
 printf ("\nValue of i = %d", *j) ;
 getch();
}
```

## OUTPUT

**Address of i = 65234**

**Address of i = 65234**

**Address of j = 65236**

**Value of j = 65234**

**Value of i = 5**

**Value of i = 5**

**Value of i = 5**

# POINTER EXAMPLE

- Look at following declarations,

`int *alpha ;`

`char *ch ;`

`float *s ;`

Here, alpha, ch and s are declared as pointer variables, i.e. variables capable of holding addresses

- Addresses (location nos.) are always going to be whole numbers; therefore pointers always contain whole numbers
- Declaration `float *s` does not mean that s is going to contain a floating-point value. What it means is, s is going to contain address of a floating-point value
- Similarly, `char *ch` means that ch is going to contain address of a char value

# DECLARING POINTER VARIABLES

- Syntax
  - `data_type *pointer_name;`
  - Eg: `int *p;`

# INITIALIZATION OF A POINTER VARIABLE

- Process of assigning address of a variable to a pointer variable is known as initialization

```
int quantity;
```

```
int *p; /* declaration*/
```

```
p=&quantity; /* initialization*/
```

- We can also combine initialization with declaration
- Only requirement here is that variable quantity must be declared before initialization takes place  

```
int*p=&quantity;
```

# INITIALIZATION OF A POINTER VARIABLE

- Consider following example:

```
int quantity,*p, n;
quantity =179;
p=&quantity; n=*p;
```

- Last line contains indirection operator \*. When operator \* is placed before a pointer variable in an expression, pointer returns value of variable of which pointer value is address
- That is, \*p returns value of variable quantity, because p is address of quantity. Thus value of n would be 179

# POINTER TO A POINTER (Chain of Pointers)

- It is possible to make pointer to point to another pointer
- Thus, we now have a pointer that contains another pointer's address
- Observe how variables p2 have been declared, `int x, *p1, **p2 ;`
- Here, x is an ordinary int, p1 is a pointer to an int (often called an integer pointer), whereas p2 is a pointer to an integer pointer
- Representation \*\*p2 is called **multiple indirection**



# POINTER TO A POINTER (Chain of Pointers)

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int var;
 int *ptr;
 int **pptr;
 var=3000;
 ptr=&var;
 pptr=&ptr;
 printf("Value of var=%d\n",var);
 printf("Value available at *ptr=%d\n",*ptr);
 printf("Value available at **pptr=%d\n",**pptr);
 getch();
}
```

## OUTPUT

Value of var=3000

Value available at \*ptr=3000

Value available at \*\*pptr=3000

# POINTER INCREMENTS & SCALE FACTOR

- Expression `p1++`; will cause pointer `p1` to point to next value of its type
- If `p1` is an integer pointer with an initial value say 2800, then after operation `p1=p1+1`, value of `p1` will be 2802, and not 2801
- When we increment a pointer its value is incremented by length of the data type that it points to. This length is called **scale factor**
- Following operations can be performed on a pointer:

# POINTER INCREMENTS & SCALE FACTOR

- Addition of a number to a pointer

```
int i = 4, *j, *k ;j
```

```
= &i;
```

```
j = j + 1;j
```

```
= j + 9;k =
```

```
j + 3;
```

- Subtraction of a number from a pointer

```
int i = 4, *j, *k ;j
```

```
= &i;
```

```
j = j - 2;
```

```
j = j - 5;
```

```
k = j - 6;
```

# Subtraction of one pointer from another

```
main ()
{
 int arr[] = { 10, 20, 30, 45, 67, 56, 74 } ;
 int *i, *j ;
 i = &arr[1] ;
 j = &arr[5] ;
 printf ("%d %d", j - i, *j - *i) ;
}
```

**OUTPUT**

8 36

# Comparison of two pointer variables

```
main ()
{
 int arr[] = { 10, 20, 36, 72, 45, 36 } ;
 int *j, *k ;
 j = &arr [4];
 k = (arr + 4) ;

 if (j == k)
 printf ("The two pointers point to the same location") ;
 else
 printf ("The two pointers do not point to the same location") ;
}
```

## OUTPUT

**The two pointers point to the same location**

# Comparison of two pointer variables

- Do not attempt following operations on pointers... they would never work out
  - (a) Addition of two pointers
  - (b) Multiplication of a pointer with a constant
  - (c) Division of a pointer with a constant

# POINTER & ARRAY

- Array name gives address of first element of array

```
#include <stdio.h>
void main()
{
 int arr[] = {10, 20, 30, 40, 50, 60};
 int *ptr = arr; // Assigns address of array to ptr
 printf("Value of first element is %d", *ptr);
 getch();
}
```

## OUTPUT

**Value of first element is 10**

# POINTER & ARRAY

```
#include <stdio.h>
void main()
{
 int arr[] = {10, 20, 30, 40, 50, 60};
 int *ptr = arr;
 printf("arr[2] = %d\n", arr[2]);
 printf("*(arr + 2) = %d\n", *(arr + 2));
 printf("*(ptr + 2) = %d\n", *(ptr + 2));
 getch ();
}
```

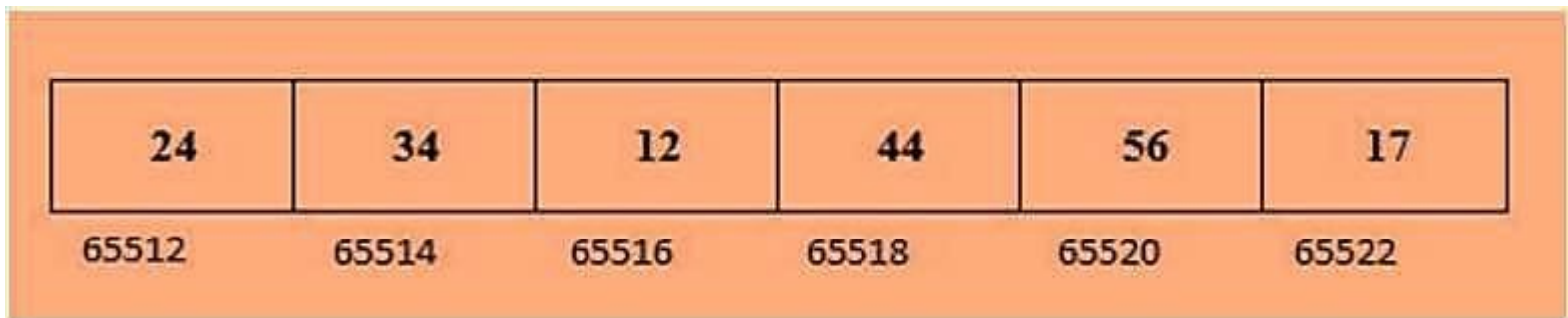
## OUTPUT

**Arr[2] = 30**  
**\*(arr + 2) = 30**  
**\*(ptr + 2) = 30**



# POINTER & ARRAY

- Suppose we have an array  
`num [ ] = {24, 34, 12, 44, 56, 17}`



# POINTER & ARRAY

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int num[] = { 24, 34, 12, 44, 56, 17 } ;
 int i, *j ;
 j = &num [0] ; /* assign address of zeroth element */
 for (i = 0 ; i <= 5 ; i++)
 {
 printf ("\naddress = %u ", j) ;
 printf ("element = %d", *j) ;
 j++; /* increment pointer to point to next location */
 }
 getch();
}
```

## OUTPUT

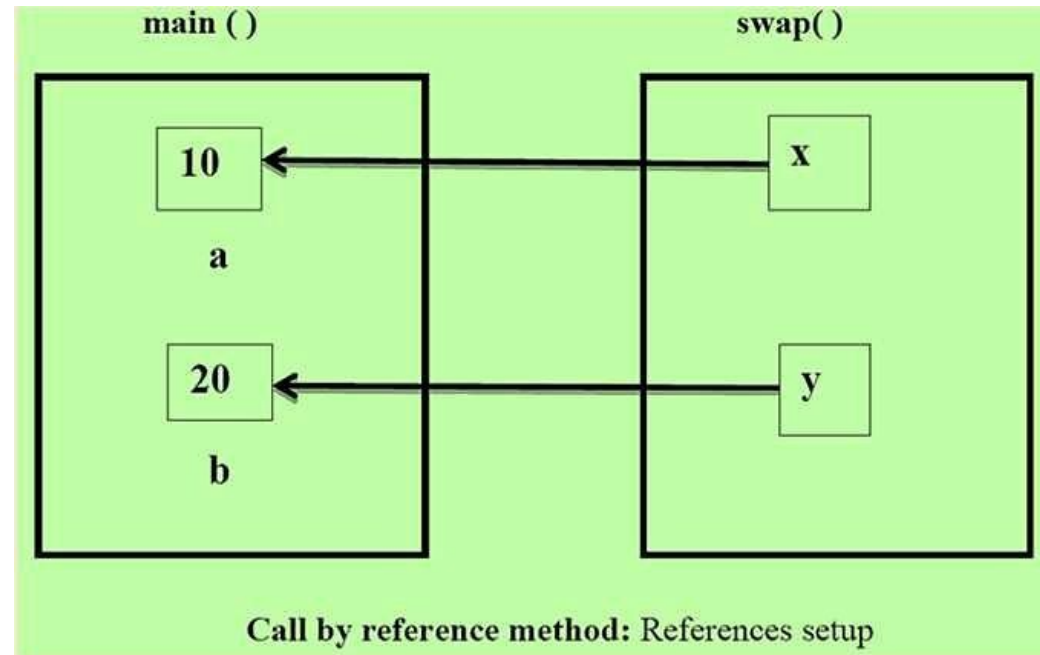
```
address = 65512 element = 24
address = 65514 element = 34
address = 65516 element = 12
address = 65518 element = 44
address = 65520 element = 56
address = 65522 element = 17
```

# PASS BY REFERENCE (CALL BY REFERENCE)

- In this method **addresses of actual arguments in calling function are copied into formal arguments of called function**
- That is **same variables value can be accessed by any of two names**: original variables name and reference name
- We are actually passes address
- Note that this program manages to exchange values of a and b using their addresses stored in x and y
- That is, **any change in formal parameter is reflected back to actual parameters**
- Usually in C programming we make a call by value

# PASS BY REFERENCE (CALL BY REFERENCE)

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



**OUTPUT**

**a = 20 b = 10**

# NULL POINTER

- It is always a good practice to assign a null value to a pointer variable in case you do not have exact address to be assigned
- This is done at time of variable declaration
- A pointer that is assigned NULL is called a null pointer
- Null pointer is a constant with a value of zero defined in several standard libraries

# NULL POINTER

Eg: `#include<stdio.h>`  
`int main()`  
`{`  
`int *ptr=NULL;`  
`printf("The value of ptr is %x\n",ptr);`  
`return 0;`  
`}`

When this code is compiled and executed, it produces the following result:

**The value of ptr is 0**

# FILE MANAGEMENT

- A file represents a sequence of bytes on disk where a group of related data is stored
- File is created for permanent storage of data
- It is a readymade structure
- A file is a place on disk where a group of related data is stored

# FILE OPERATIONS

1. Creation of a new file
2. Writing to a file
3. Opening an existing file
4. Reading from a file
5. Moving to a specific location in a file (seeking)
6. Closing a file



# FUNCTIONS FOR FILE HANDLING

| Function  | Description                                        |
|-----------|----------------------------------------------------|
| fopen()   | opens new or existing file                         |
| fprintf() | write data into the file                           |
| fscanf()  | reads data from the file                           |
| fputc()   | writes a character into the file                   |
| fgetc()   | reads a character from file                        |
| fclose()  | closes the file                                    |
| fseek()   | sets the file pointer to given position            |
| fputw()   | writes an integer to file                          |
| fgetw()   | reads an integer from file                         |
| ftell()   | returns current position                           |
| rewind()  | sets the file pointer to the beginning of the file |

# General format for declaring and opening a file

```
FILE *fp;
```

```
fp=fopen ("filename","mode");
```

- fp is a pointer to data type FILE
- This pointer contains all information about file

# Trouble in Opening a File

- It is important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to file
- If file opening fails due to any of several reasons, `fopen( )` function returns a value `NULL`
- `NULL` is defined in “`stdio.h`” as  
`#define NULL 0`

# Trouble in Opening a File

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fp ;
 fp = fopen ("ONE.C", "r") ;
 if (fp == NULL)
 {
 puts ("cannot open file") ;
 exit() ;
 }
 getch();
}
```

# Program to read a file and display its contents on screen

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fp;
 char ch;
 fp = fopen ("ONE.C", "r") ;
 while (1)
 {
 ch = fgetc (fp) ;
 if (ch == EOF)
 break ;
 printf ("%c", ch) ;
 }
 fclose (fp) ;
 getch();
}
```

# COUNTING CHARACTERS, TABS and SPACES

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fp ;
 char ch ;
 int nol = 0, not = 0, nob = 0, noc = 0 ;
 fp = fopen ("PR1.C", "r") ;
 while (1)
 {
 ch = fgetc (fp) ;
 if (ch == EOF)
 break ;
 noc++ ;
 }
}
```

# COUNTING CHARACTERS, TABS and SPACES

```
 if (ch == ' ')
 nob++;
 if (ch == '\n')
 nol++;
 if (ch == '\t')
 not++;
}
fclose (fp);
printf ("\nNumber of characters = %d", noc);
printf ("\nNumber of blanks = %d", nob);
printf ("\nNumber of tabs = %d", not);
printf ("\nNumber of lines = %d", nol);
}
```

# FILE-COPY PROGRAM

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fs, *ft ;
 char ch ;
 fs = fopen ("pr1.c", "r") ;
 if (fs == NULL)
 {
 puts ("Cannot open source file") ;
 exit() ;
 }
 ft = fopen ("pr2.c", "w") ;
```



# FILE-COPY PROGRAM

```
if (ft == NULL)
{
 puts ("Cannot open target file");
 fclose (fs);
 exit();
}
while (1)
{
 ch = fgetc (fs);
 if (ch == EOF)
 break ;
 else
 fputc (ch, ft);
}
fclose (fs);
fclose (ft);
}
```

# fprintf ( ) and fscanf ( )

- fprintf and fscanf can handle a group of mixed data simultaneously
- General form is

`fprintf ( fp, "control string", list);`

fp is a file pointer associated with a file that has been opened for writing

Eg: `fprintf (f1, "%s%d%f", name, age, 7.5);`

- General format of fscanf is `fscanf`

`(fp, "control string", list);`list

means address of variables

Eg: `fscanf(f2, "%s%d", item, &quantity);`

# FILE OPENING MODE

| Mode      | Meaning             | Description                                                                                                                        |
|-----------|---------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>r</b>  | Read                | Only reading possible. Not create file if not exist                                                                                |
| <b>w</b>  | Write               | Only writing possible. Create file if not exist otherwise erase the old content of file and open as a blank file                   |
| <b>a</b>  | Append              | Only writing possible. Create file if not exist, otherwise open file and write from the end of file (do not erase the old content) |
| <b>r+</b> | Reading + Writing   | R & W possible. Create file if not exist. Overwriting existing data. Used for modifying content                                    |
| <b>w+</b> | Reading + Writing   | R & W possible. Create file if not exist. Erase old content.                                                                       |
| <b>a+</b> | Reading + Appending | R & W possible. Create file if not exist. Append content at the end of file                                                        |

# Storing Employee information

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fptr;
 int id;
 char name[30];
 float salary;
 fptr = fopen("emp.txt", "w+");
 if (fptr == NULL)
 {
 printf("File does not exists \n");
 return;
 }
}
```

# Storing Employee information

```
printf("Enter the id\n");
scanf("%d", &id);
fprintf(fptr, "Id= %d\n", id);
printf("Enter the name \n");
scanf("%s", name);
fprintf(fptr, "Name= %s\n", name);
printf("Enter the salary\n");
scanf("%f", &salary);
fprintf(fptr, "Salary= %.2f\n", salary);
fclose(fptr);
}
```

# feof()

- C provides feof() which returns non-zero value only if end of file has reached, otherwise it returns 0
- For example, consider following C program to print contents of file test.txt on screen
- In program, returned value of getc() is compared with EOF first
- Then there is another check using feof()
- By putting this check, we make sure that program prints “End of file reached” only if end of file is reached
- If getc() returns EOF due to any other reason, then program prints “Something went wrong”

# feof()

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

```
int main()
```

```
{
```

```
 FILE *fp = fopen("test.txt", "r");
```

```
 int ch = getc(fp);
```

```
 while (ch != EOF)
```

```
 {
```

```
 /* display contents of file on screen */
```

```
 putchar(ch);
```

```
 ch = getc(fp);
```

```
 }
```

```
 if (feof(fp))
```

```
 printf("\n End of file reached.");
```

```
 else
```

```
 printf("\n Something went wrong.");
```

```
 fclose(fp);
```

```
 return 0;
```

```
}
```



# fseek() function

- fseek() function is used to set file pointer to the specified offset
- It is used to write data into file at desired location
- Syntax: `int fseek(FILE *stream, long int offset, int whence)`
- There are 3 constants used in fseek()
- Function for whence:
  - SEEK\_SET - beginning of file
  - SEEK\_CUR - current position
  - SEEK\_END - end of file



# fseek() function

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fp;
 fp = fopen("myfile.txt","w+");
 fputs("This is new", fp);
 fseek(fp, 7, SEEK_SET);
 fputs("eduline", fp);
 fclose(fp);
}
```

## OUTPUT

This is eduline

# rewind()

- `rewind()` function sets file pointer at beginning of stream
- It is useful if you have to use stream many times
- Syntax: `void rewind(FILE *stream)`

# rewind()

```
include <stdio.h>
include <conio.h>
void main()
{
 FILE *fp;
 char c;
 clrscr();
 fp=fopen("file.txt","r");
 while((c=fgetc(fp))!=EOF)
 {
 printf("%c",c);
 }
}
```

```
rewind(fp); //moves the file pointer at beginning of the file
while((c=fgetc(fp))!=EOF)
{
 printf("%c",c);
}
fclose(fp);
getch();
}
```

**OUTPUT -** This is simple This is simple

# ftell()

- ftell() in C is used to find out position of file pointer in file with respect to starting of file
- Syntax of ftell() is: `ftell(FILE *pointer)`
- Consider below C program
- File taken in example contains following data:  
“Someone over there is calling you. We are going for work. Take care of yourself.” (without quotes)
- When fscanf statement is executed word “Someone” is stored in string and pointer is moved beyond “Someone”
- Therefore ftell(fp) returns 7 as length of “someone”

# ftell()

```
// C program to demonstrate use of ftel()
#include<stdio.h>

int main()
{
 /* Opening file in read mode */
 FILE *fp = fopen("test.txt","r");

 /* Reading first string */
 char string[20];
 fscanf(fp,"%s",string);

 /* Printing position of file pointer */
 printf("%ld", ftell(fp));
 return 0;
}
```

**Output :**  
**7**

**END....**