

## UNIT I: Introduction to Programming and Problem Solving

History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program Algorithms, flowcharts (Using Dia Tool), pseudo code. Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operations, Type Conversion, and Casting.

**Problem solving techniques:** Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms.

---

### HISTORY OF COMPUTERS:

First Personal Computer (Altair 8800, 1975): The Altair 8800, designed by Ed Roberts, was one of the first commercially successful personal computers. It featured an Intel 8080 microprocessor and was sold as a kit for enthusiasts.

BM PC (1981): IBM introduced the IBM Personal Computer (IBM PC), which became a standard in the personal computer industry. Its open architecture and use of the MS-DOS operating system contributed to its widespread adoption.

Graphical User Interface (GUI, 1980s): Xerox PARC developed the concept of the graphical user interface, which was later popularized by Apple's Macintosh and Microsoft's Windows operating systems.

Internet (1960s-1980s): The development of ARPANET, the precursor to the internet, began in the late 1960s. It evolved into a global network that revolutionized communication and information exchange.

Smartphones and Mobile Computing (2000s): The introduction of smart phones, such as the iPhone in 2007, transformed computing by making powerful computing devices portable and accessible to a mass audience.

Cloud Computing (2000s): Cloud computing emerged as a paradigm where computing resources are delivered as a service over the internet. This revolutionized how businesses and individuals access and utilize computing power and storage.

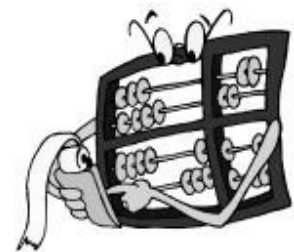
Quantum Computing (Ongoing): Quantum computing is an emerging field that harnesses the principles of quantum mechanics to perform complex calculations at speeds far beyond classical computers. It has the potential to revolutionize various industries but is still in its early stages.

The history of computers is a story of constant innovation and evolution, with each generation of technology building upon the successes and limitations of the previous one. Today, computers are an integral part of modern life, impacting everything from communication and entertainment to science and business.

**By completing this module, you will be able to understand and learn the following:**

- **The history of Computers**
- **Generations of Computing**
- **Types of Computers**
- **Analog, Digital & Hybrid Computers**

The invention process of the computer started around 3000 years ago. The computer started out as an "Abacus". An 'Abacus' is a rack made of wood with two wires running parallel to each other. On the wires there are beads. By moving the beads anyone can solve simple math problems. Next, there was the 'Astrolabe', used for navigating.



The first digital computer was invented in 1642 by Blaise Pascal. It consisted of numbers entered in dials but, it could only add. However in 1671, a computer was invented that was eventually built in 1694. The man to credit for this invention is Gottfried Wilhelm von Leibniz. Unlike Pascal's computer, Leibniz's could add and multiply.



**“A computer is a machine that manipulates data according to a set of instructions.”**

### **Time line of the history of computers**

**3000 B.C.:** The abacus was invented in Babylon, marking one of the earliest known tools for calculation.

**1800 B.C.:** Babylonians developed algorithms for solving number problems, laying foundational concepts for mathematical computation.

**500 B.C.:** Egyptians created a bead and wire abacus, enhancing calculation methods with a more practical tool.

**200 B.C.:** Japanese started using computing trays.

**1617:** John Napier, a Scottish inventor, demonstrated how to divide by subtraction and how to multiply by addition.

**1624:** Wilhelm Schickard invented the first four-function calculator-clock at Heidelberg University.

**1642:** Blaise Pascal invented the first numerical calculating machines in Paris.

**1780:** Benjamin Franklin discovered electricity.

**1876:** Alexander Graham Bell invented the telephone.

**1886:** William Burroughs developed the first commercial mechanical adding machine.

**1896:** Hollerith constructed a sorting machine.

**1925:** Vannevar Bush built the large-scale analog calculator, the differential analyzer, at MIT.

**1927:** The first public radio-telephone became operational between London and New York.

**1931:** Konrad Zuse built the Z1, the first calculator in Germany.

**1936:** Alan M. Turing defined a machine capable of computing any calculatable function.

**1937:** George Stibitz built the first binary calculator at Bell Telephone Laboratories.

**1938:** Hewlett-Packard Company began making electric equipment.

**1948:** IBM introduced the 604 electronic calculators.

**1953:** Remington-Rand developed the first high-speed printer.

**1958:** NEC, Japan, developed the first electronic computer.

**1960:** Removable disks appeared for the first time.

**1972:** Intel introduced an 8-bit microprocessor.

**1976:** Perkin-Elmer and Gould SEL introduced superminicomputers.

**1977:** The Apple II personal computer was introduced.

### **Father of Computing- Charles Babbage**

Charles Babbage was an English inventor and mathematician who, in the 1800's, believed he could build a computing machine. In 1827, after convincing the British government to finance his project, he worked for years on his Difference Engine, a device intended for the production of tables. While he produced

prototypes of portions of the Difference Engine, eventually he gave up. In 1854, he decided to build an Analytical Engine, which was also left unfinished. However, his proposals for mechanical computers predated the modern reinvention of computers by almost a century. Because of this accomplishment, Charles Babbage has earned his place in history as the "Father of Computing."

### **Generations of computing**

Generation in computer terminology is a change in technology a computer is/was being used. Initially, the generation term was used to distinguish between varying hardware technologies. But nowadays, generation includes both hardware and software, which together make up an entire computer system.

S.N.	Generation & Description
1.	<b>First Generation</b> The period of first generation: 1946-1959. Vacuum tube based.
2.	<b>Second Generation</b> The period of second generation: 1959-1965. Transistor based.
3.	<b>Third Generation</b> The period of third generation: 1965-1971. Integrated Circuit based.
4.	<b>Fourth Generation</b> The period of fourth generation: 1971-1980. VLSI micro processor based.
5.	<b>Fifth Generation</b> The period of fifth generation: 1980-onwards. ULSI microprocessor based

#### **First generation:**

The period of first generation was 1946-1959. The computers of first generation used vacuum tubes as the basic components for memory and circuitry for CPU (Central Processing Unit). These tubes, like electric bulbs, produced a lot of heat and were prone to frequent fusing of the installations, therefore, were very expensive and could be afforded only by very large organizations. In this generation mainly batch processing operating system were used. Punched cards, paper tape, and magnetic tape were used as input and output devices. The computers in this generation used machine code as programming language.



The main features of first generation are:

- i Vacuum tube technology
- ii Unreliable
- iii Supported machine language only
- iv Very costly
- v Generated lot of heat
- vi Slow input and output devices
- vii Huge size
- viii Need of A.C.
- ix Non-portable
- x Consumed lot of electricity

**Some computers of this generation were:**

1. ENIAC

2. EDVAC
3. UNIVAC
4. IBM-701
5. IBM-650

### **Second generation:**

The period of second generation was 1959-1965. In this generation transistors were used that were cheaper, consumed less power, more compact in size, more reliable and faster than the first generation machines made of vacuum tubes. In this generation, magnetic cores were used as primary memory and magnetic tape and magnetic disks as secondary storage devices. In this generation assembly language and high-level programming languages like FORTRAN, COBOL was used. The computers used batch processing and multiprogramming operating system.



The main features of second generation are:

- Use of transistors
- Reliable in comparison to first generation computers
- Smaller size as compared to first generation computers
- Generated less heat as compared to first generation computers
- Consumed less electricity as compared to first generation computers
- Faster than first generation computers
- Still very costly
- A.C. needed
- Supported machine and assembly languages

Some computers of this generation were:

1. IBM 1620
2. IBM 7094

3. CDC 1604
4. CDC 3600
5. UNIVAC 1108

### **Third generation:**

The period of third generation was 1965-1971. The computers of third generation used integrated circuits (IC's) in place of transistors. A single IC has many transistors, resistors and capacitors along with the associated circuitry. The IC was invented by Jack Kilby. This development made computers smaller in size, reliable and efficient. In this generation remote processing, time-sharing, multi-programming operating system were used. High level languages (FORTRAN-II TO IV, COBOL, PASCAL PL/1, BASIC, ALGOL-68 etc.) were used during this generation.



The main features of third generation are:

- 1 IC used
- 2 More reliable in comparison to previous two generations
- 3 Smaller size
- 4 Generated less heat
- 5 Faster
- 6 Lesser maintenance
- 7 Still costly
- 8 A.C needed
- 9 Consumed lesser electricity
- 10 Supported high-level language

Some computers of this generation were:

1. IBM-360 series

2. Honeywell-6000 series
3. PDP(Personal Data Processor)
4. IBM-370/168
5. TDC-316

#### **Fourth generation:**

The period of fourth generation was 1971-1980. The computers of fourth generation used Very Large Scale Integrated (VLSI) circuits. VLSI circuits having about 5000 transistors and other circuit elements and their associated circuits on a single chip made it possible to have microcomputers of fourth generation. Fourth generation computers became more powerful, compact, reliable, and affordable. As a result, it gave rise to personal computer (PC) revolution. In this generation time sharing, real time, networks, distributed operating system were used. All the high-level languages like C, C++, DBASE etc., were used in this generation.



The main features of fourth generation are:

- 1 VLSI technology used
- 2 Very cheap
- 3 Portable and reliable
- 4 Use of PC's
- 5 Very small size
- 6 Pipeline processing
- 7 No A.C. needed
- 8 Concept of internet was introduced
- 9 Great developments in the fields of networks
- 10 Computers became easily available

Some computers of this generation were:

- I) DEC 10



II) STAR 1000

III) PDP 11

IV) CRAY-1 (Super Computer)

V) CRAY-X-MP (Super Computer)

### **Fifth generation:**

The period of fifth generation is 1980-till date. In the fifth generation, the VLSI technology became ULSI (Ultra Large Scale Integration) technology, resulting in the production of microprocessor chips having ten million electronic components. This generation is based on parallel processing hardware and AI (Artificial Intelligence) software. AI is an emerging branch in computer science, which interprets means and method of making computers think like human beings. All the high-level languages like C and C++, Java, .Net etc., are used in this generation.

*AI includes:*

- Robotics
- Neural Networks
- Game Playing
- Development of expert systems to make decisions in real life situations.
- Natural language understanding and generation



The main features of fifth generation are:

- a) ULSI technology
- b) Development of true artificial intelligence
- c) Development of Natural language processing
- d) Advancement in Parallel Processing
- e) Advancement in Superconductor technology

- f) More user friendly interfaces with multimedia features
- g) Availability of very powerful and compact computers at cheaper rates

Some computer types of this generation are:

1. Desktop
2. Laptop
3. Note Book
4. Ultra Book
5. Chrome Book

### **Computer Types (or) Types of Computers (or) Classification of Computers (or) Classification based on Operating Principles:-**

Based on the operating principles, computer can be classified into one of the following types:

- Digital Computers
- Analog Computers
- Hybrid Computers

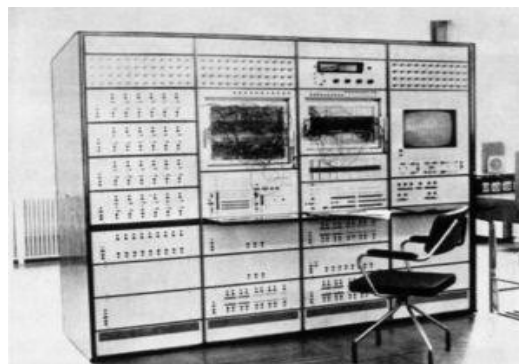
**Digital Computers:** - Operate essentially by counting. All quantities are expressed as discrete or numbers. Digital computers are useful for evaluating arithmetic expressions and manipulations of data (such as preparation of bills, ledgers, solution)



**Analog Computers:-** An analog computer is a form of computer that uses the continuously changeable aspects of physical phenomena such as electrical, mechanical, or hydraulic quantities to model the problem being solved. In contrast, digital computers represent varying quantities symbolically, as the in numerical values change.



**Hybrid Computers:-** are computers that exhibit features of analog computers and digital computers. The digital component normally serves as the controller and provides logical operations, while the analog component normally serves as a solver of differential equations.



### **Classification digital Computer based on size and Capability**

Based on size and capability, computers are broadly classified into **Micro Computers (Personal Computer)**

A microcomputer is the smallest general purpose processing system. The old erpc started 8bit processor with speed of 3.7MB and current pc 64 bit processor with speed of 4.66 GB.

Examples: **IBM PCs, APPLE computers**

Microcomputer can be classified into 2 types:

Desktops

## Portables

The difference is portables can be used while travelling where as desktops computers cannot be carried around.

**The different portable computers are:-**

Laptop

Notebooks

Palmtop (hand held)

Wearable computers

**Laptop:-** This computer is similar to a desktop computers but the size is smaller. They are expensive than desktop. The weight of laptop is around 3 to 5kg.



**Notebook:** - These computers are as powerful as desktop but size of these computers are comparatively smaller than laptop and desktop. They weigh 2 to 3 kg. They are more costly than laptop.



**Palmtop (Hand held):-** They are also called as personal Digital Assistant (PDA). These computers are small in size. They can be held in hands. It is capable of doing word processing, spread sheets and hand writing recognition, game playing, faxing and paging. These computers are not as powerful as desktop computers.

Ex:- 3Com Palm V.



**Wearable computer:** - The size of this computer is very small so that it can be worn on the body. It has smaller processing power. It is used in the field of medicine.

For example A **sulin meter**, commonly known as a **blood glucose meter**, is a device used to measure the level of insulin in the blood indirectly by monitoring blood glucose levels.



**Workstations:-** It is used in large, high-resolution graphics screen built in network support, Engineering applications (CAD/CAM), software development desktop publishing

Ex: Unix and windows NT.

**Minicomputer:-** A mini computer is a medium-sized computer. That is more powerful than a microcomputer. These computers are usually designed to serve multiple users simultaneously (Parallel Processing). They are more expensive than micro computers.

Examples: Digital Alpha, Sun Ultra.



**Mainframe (Enterprise) computers:** - Computers with large storage capacities and very high speed of processing (compared to mini-or micro computers) are known as mainframe computers. They support a large number of terminals for simultaneous use by a number of users like ATM transactions. They are also used as central host computers in distributed at a processing system.

Examples:- **IBM370,S/390.**



**Supercomputer:**-Super computers have extremely large storage capacity and computing speeds which are many times faster than other computers. A supercomputer is measured in terms of tens of millions Instructions per second (mips), an operation is made up of numerous instructions. The super computer is mainly used for large scale numerical problems in scientific and engineering disciplines such as Weather analysis.

Examples:-**IBM Deep Blue**



### Classification based on number of micro processors

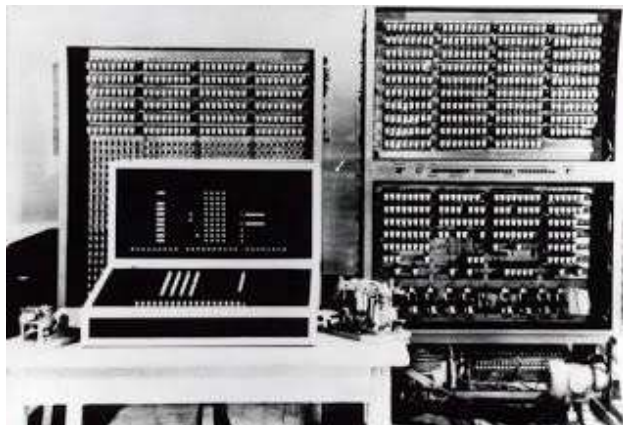
Based on the number of micro processors, computers can be classified into Sequential computers and Parallel computers.

**Sequential computers:** -Any task complete in sequential computers is with one micro computer only. Most of the computers (today) we see are sequential computers wherein any task is completed sequentially instruction after instruction from the beginning to the end.

**Parallel computers:** - The parallel computer is relatively fast. New types of computers that use a large number of processors. The processors perform different tasks independently and simultaneously thus improving the speed of execution of complex programs dramatically. Parallel computers match the speed of super computer satisfaction of the cost.

### Classification based on word-length

A binary digit is called “**BIT**”. A word is a group of bits which is fixed for a computer. The number of bits in a word (or word length) determines the representation of all characters in these many bits. Word length leis in the range from 16-bit to 64-bitsf or most computers of today.



### Classification based on number of users

Based on number of users, computers are classified into:-

**Single User:**-Only one user can use the resource at any time.





**Multi User:-**A single computer shared by a number of users at any time.



**Network:-** A number of interconnected autonomous computers shared by multiple users at any time.





## **COMPUTER TYPES**

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information. List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

### **Personal computers:-**

This is the most common type found in homes, schools, and business offices. It includes desktop computers with processing and storage units, along with various input and output devices.

**Notebook computers:-** These are compact and portable versions of PCs

**Work stations:** - Workstations have high-resolution input/output (I/O) graphics capabilities while maintaining the same dimensions as desktop computers. They are commonly used in engineering applications for interactive design work

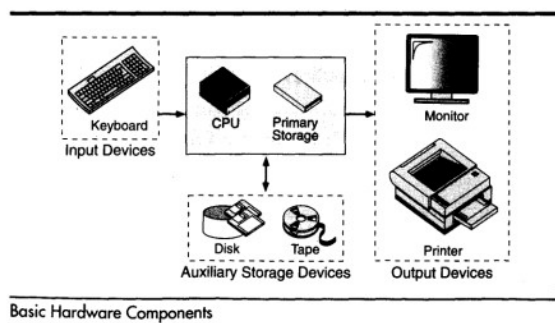
**Enterprise systems:** -These are used for business data processing in medium to large corporations that require significantly more computing power and storage capacity than workstations. Servers connected to the internet have become a dominant worldwide source of all types of information..

**Super computers:** - These are used for large-scale numerical calculations required in applications such as weather forecasting.

### **Basic organization of a computer:**

#### **Computers have two kinds of components:**

**Hardware:** It consists of the physical devices of a computer, including the CPU, memory, bus, and storage devices



**Software:** It consists of the programs it has, including the operating system, applications, and utilities.

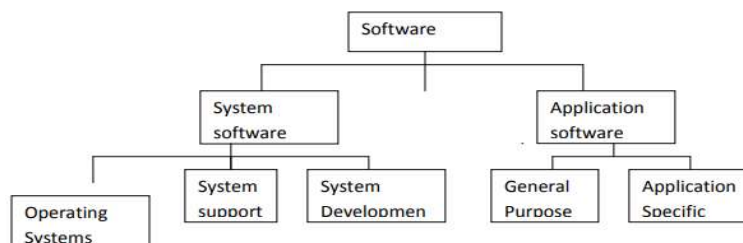
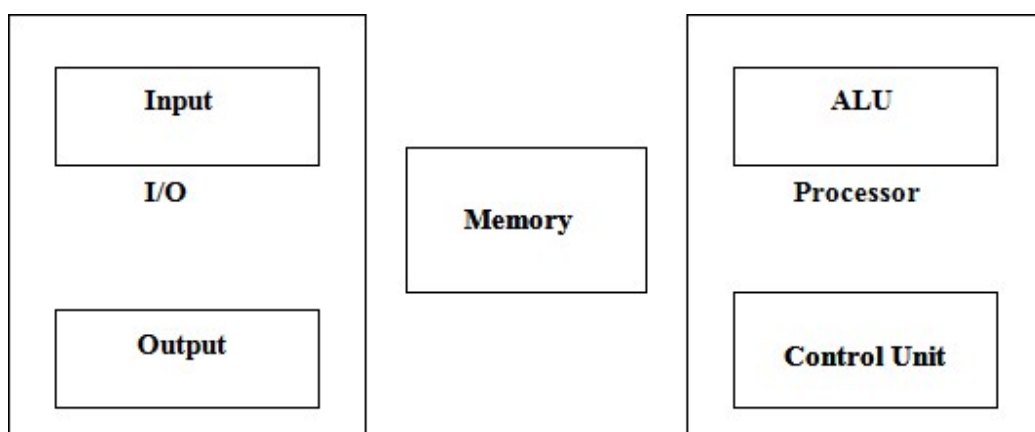


Fig: Types of software

Activat  
Go to Se

## FUNCTIONAL UNIT

A computer consists of five functionally independent main parts: input, memory, arithmetic logic unit (ALU), output, and control unit.

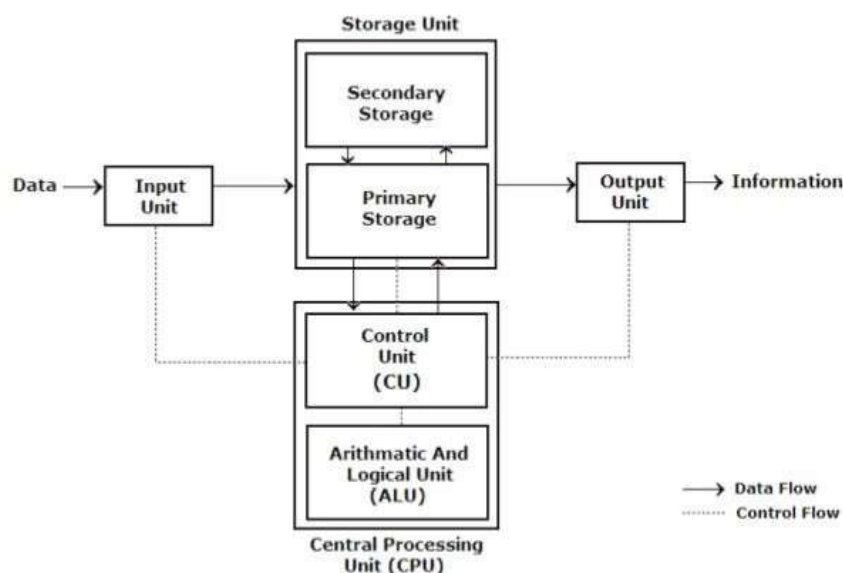


Functional units of computer

An input device accepts coded information as a source program, which is typically written in a high-level language. This information is either stored in memory or immediately used by the processor to perform the desired operations. The program stored in memory determines the processing steps. Essentially, the computer converts one source program into an object program, i.e., into machine language..

Finally, the results are sent to the outside world through an output device. All of these actions are coordinated by the control unit.

## Block diagram of computer



### Input unit:-

The source program, high-level language program, coded information, or simply data is fed to a computer through input devices, with the keyboard being the most common type. Whenever a key is pressed, the corresponding word or number is translated into its equivalent binary code, which is then sent over a cable to either memory or the processor.

Other input devices include joysticks, trackballs, mice, and scanners

**Memory Unit:** Its function is to store programs and data. It is basically divided into two types:

1. **Primary Memory**
2. **Secondary Memory**

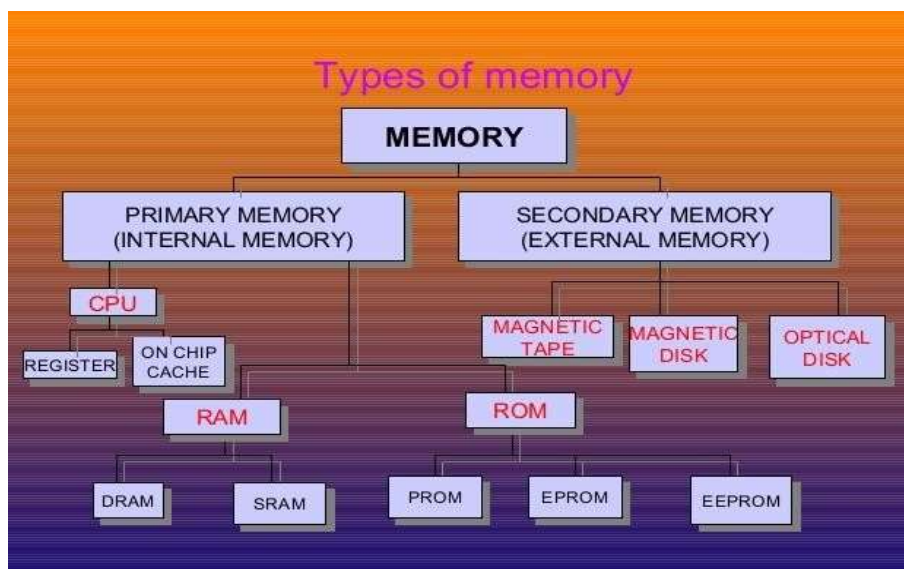
### **Word:**

In computer architecture, a word is a unit of data of a defined bit length that can be addressed and moved between storage and the computer processor. The defined bit length of a word is typically equivalent to the width of the computer's data bus, allowing a word to be moved in a single operation from storage to a processor register. For any computer architecture with an eight-bit byte, the word will be some multiple of eight bits.

In IBM's System/360 architecture, a word is 32 bits, or four contiguous eight-bit bytes. In Intel's PC processor architecture, a word is 16 bits, or two contiguous eight-bit bytes. A word can contain a computer instruction, a storage address, or application data that is to be manipulated (for example, added to the data in another word space).

The number of bits in each word is known as word length. Word length refers to the number of bits processed by the CPU at one time. In modern general-purpose computers, word size can range from 16 bits to 64 bits.

The time required to access one word is called memory access time. The small, fast RAM units are known as caches. They are tightly coupled with the processor and are often contained on the same IC chip to achieve high performance.



## 1. Primary Memory

Primary memory is exclusively associated with the processor and operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information. These bits are processed in groups of fixed size called words.

To provide easy access to a word in memory, each word location is associated with a distinct address. These addresses are numbers that identify specific memory locations.

The number of bits in each word is called the word length of the computer. Programs must reside in memory during execution, and instructions and data can be written to or read from memory under the control of the processor. Memory from which any location can be reached in a short and fixed amount of time after specifying its address is known as random-access memory (RAM).

The time required to access one word is referred to as memory access time. Memory that is only readable by the user, with contents that cannot be altered, is called read-only memory (ROM), which typically contains the operating system.

Caches are small, fast RAM units tightly coupled with the processor and often contained on the same IC chip to achieve high performance. Although primary storage is essential, it tends to be expensive.

## 2. Secondary Memory

Secondary memory is used for storing large amounts of data and programs, particularly information that is accessed infrequently.

**Examples:** Magnetic disks and tapes, optical disks (e.g., CD-ROMs), and floppy disks.

### Memory Measurement Units:

Computer Memory Measurement Units

SYMBOL	FULL FORM	QUANTITY
1 BIT	BINARY DIGIT	1 CELL, BINARY 0 OR 1
4 BITS	NIBBLE	1/2 BYTE
8 BITS	BYTE	1 BYTE
1024 BYTE	KILOBYTE	1 KILOBYTE
1024 KILOBYTE	MEGABYTE	1 MEGABYTE
1024 MEGABYTE	GIGABYTE	1 GIGABYTE
1024 GIGABYTE	TERABYTE	1 TERABYTE
1024 TERABYTE	PETABYTE	1 PETABYTE
1024 PETABYTE	HEXABYTE	1 HEXABYTE
1024 HEXABYTE	ZEETABYTE	1 ZEETABYTE

## Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) executes most of the computer's operations, including addition, subtraction, division, and multiplication. The operands are brought into the ALU from memory and stored in high-speed storage elements called registers. The operation is then performed in the required sequence according to the instructions.

The control unit and the ALU operate many times faster than other devices connected to a computer system, allowing a single processor to manage multiple external devices, such as keyboards, displays, magnetic and optical disks, sensors, and other mechanical controllers.

## Output Unit

The output unit serves as the counterpart to the input unit. Its basic function is to send the processed results to the outside world.

**Examples:** Printers, speakers, monitors, etc.

## Control Unit

The control unit acts as the nerve center of the computer. It sends signals to other units and monitors their states. The actual timing signals that govern the transfer of data between the input unit, processor, memory, and output unit are generated by the control unit.

## Basic Operational Concepts

To perform a given task, an appropriate program consisting of a list of instructions is stored in memory. Individual instructions are brought from memory into the processor, which executes the specified operations. Data to be processed is also stored in memory.

### **Example:** **Add LOCA, R0**

This instruction adds the operand at memory location LOCA to the operand in register R0 and places the sum in R0. The execution of this instruction involves several steps:

1. The instruction is fetched from memory into the processor.
2. The operand at LOCA is fetched and added to the contents of R0.
3. Finally, the resulting sum is stored in register R0.

The preceding add instruction combines a memory access operation with an ALU operation. In some other types of computers, these two operations are performed by separate instructions for performance reasons.

**Example:**  
**Load LOCA, R1**  
**Add R1, R0**

Transfers between the memory and the processor are initiated by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data is then transferred to or from memory.

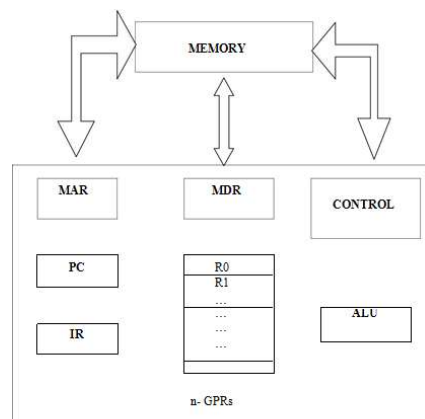


Fig b : Connections between the processor and the memory

The above figure illustrates how memory and the processor can be connected. In addition to the ALU and the control circuitry, the processor contains a number of registers used for various purposes.

## Register

A register is a special, high-speed storage area within the CPU. All data must be represented in a register before it can be processed. For example, if two numbers are to be multiplied, both numbers must be in registers, and the result is also stored in a register. (A register can contain the address of a memory location where data is stored rather than the actual data itself.)

The number of registers a CPU has and the sizes of each register (in bits) help determine the power and speed of the CPU. For instance, a 32-bit CPU has registers that are 32 bits wide, allowing each CPU instruction to manipulate 32 bits of data. In high-level languages, the compiler is responsible for translating high-level operations into low-level operations that access these registers.

## Types of registers:

**Input Register**-8 bits: Holds the user input

**Output Register**-8 bits: Holds the output

**Address Register**-12 bits: Holds address of memory locations.

**Program counter**-12 bits: Holds address of next instructions which is to be executed

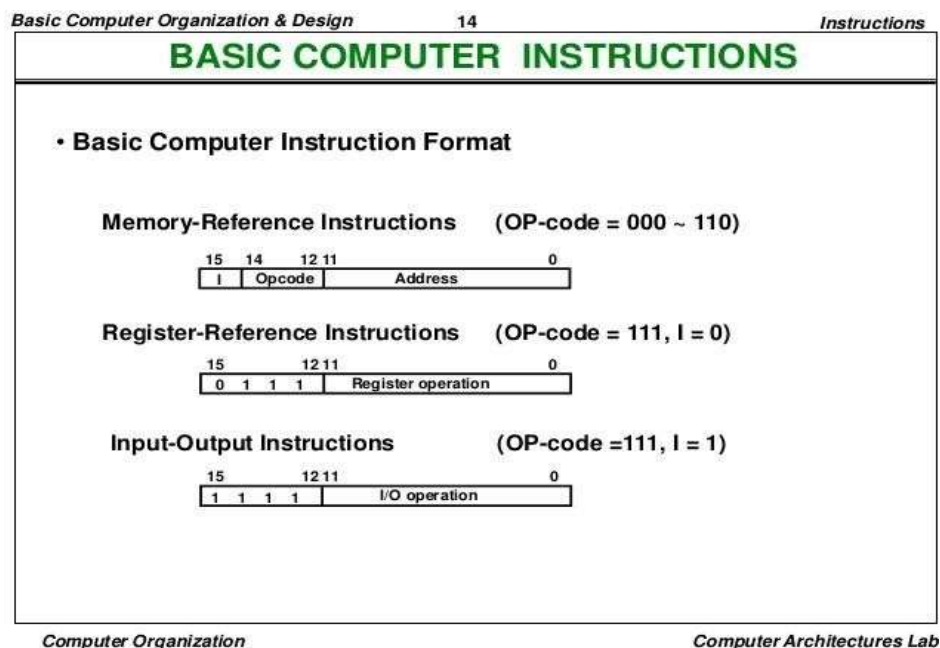
**Data Register**-16 bits: Holds the data of memory location.

**Accumulator**-16 bits: Special-purpose register used to store intermediate results of arithmetic and logic operations.

**Temporary Register**-16 bits: Holds temporary data.

**Instruction Register**- 16 bits: Holds instruction read from memory.

### Basic Computer Instructions:-



Computer instructions are the fundamental building blocks of a machine language program. They are also known as macro operations because each instruction consists of sequences of micro operations.

Each instruction starts a series of micro operations that fetch operands from registers or memory, perform arithmetic, logic, or shift operations, and store results back in registers or memory.

Instructions are encoded as binary codes. Each instruction code includes an operation code (op-code) that indicates the main action of the instruction (e.g., add, subtract, move, input). The



number of bits used for the opcode determines how many different instructions the architecture can support.

In addition to the opcode, many instructions also have one or more operands, which show where the data needed for the operation is located in registers or memory. For example, an add instruction requires two operands, while a not instruction requires one.

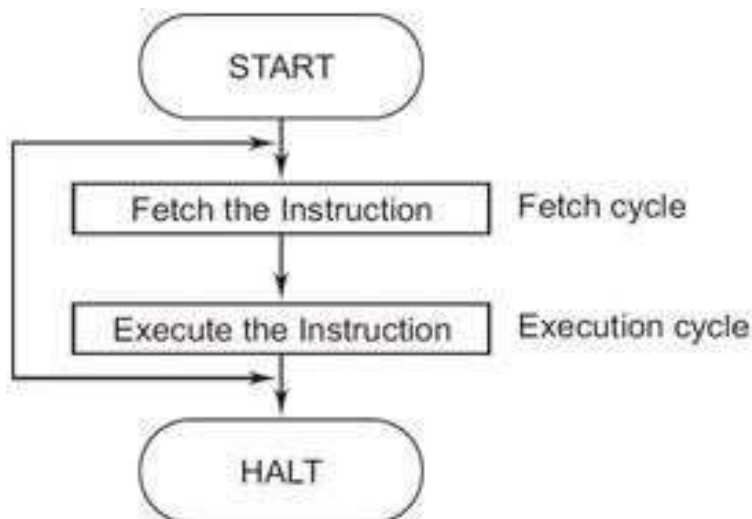
Here's a simple layout of an instruction:

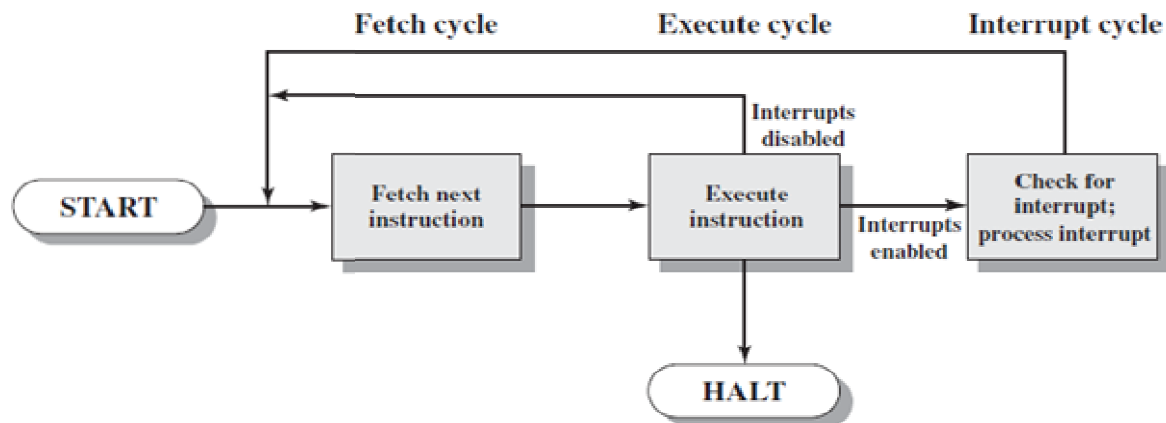
```
diff
Copy code
+-----+
| Opcode | Operand | Operand |
+-----+
```

The opcode and operands are usually represented as unsigned binary numbers to save space. For instance, a 4-bit opcode can represent up to 16 different operations.

The control unit is responsible for decoding the opcode and operand bits in the instruction register. It then generates control signals to coordinate all other hardware in the CPU to carry out the micro operations needed for the instruction.

### INSTRUCTION CYCLE:





Instruction Cycle with Interrupts

**1. Program Counter (PC):** Keeps track of the memory address of the next instruction to be executed.

**2. Instruction Register (IR):** Holds the current instruction being executed.

**3. Memory Address Register (MAR):** Contains the address of the memory location to be accessed.

**4. Memory Data Register (MDR):** Holds the data to be read from or written to memory.

### Execution Steps:

#### 1. Loading the Program:

- Programs are loaded into memory through the Input / Output unit.
- Execution begins when the PC is set to the address of the first instruction.

#### 2. Fetching the Instruction:

- The address in the PC is copied to the MAR.
- A Read Control Signal is sent to access memory.
- After the memory access time, the instruction is read into the MDR.

### *3. Decoding the Instruction:*

- The instruction in the MDR is transferred to the IR for decoding and execution.

### *4. Executing the Instruction:*

- If the instruction involves arithmetic or logic operations, the required operands must be fetched.
- For each operand needed:
  - Its address is sent to the MAR.
  - A read cycle is initiated to fetch the operand into the MDR.
- The operand is then transferred from the MDR to the Arithmetic Logic Unit (ALU).

### *5. Storing Results:*

- If the result needs to be stored in memory:
  - The result is sent to the MDR.
  - The address where the result should be stored is sent to the MAR.
- A write cycle is initiated.

### *6. Updating the Program Counter:*

- After executing the instruction, the PC is incremented to point to the next instruction.

### **Handling Interrupts:**

### *1. Interrupt Signal:*

- An interrupt is a request from an I/O device for the processor's attention.

### *2. Interrupt Service:*

- The processor saves its current state to memory before handling the interrupt.
- The processor executes an Interrupt Service Routine to address the request.

### *3. Resuming Execution:*

- After the interrupt service is completed, the saved state of the processor is restored.
- The interrupted program resumes execution from where it was interrupted.

## **Program Counter:-**

Many complex components come together to make a computer system work seamlessly. One such essential element is the program counter. There is a register in a PC (program counter) processor that contains the address of the next instruction to be executed from memory.

In this article we learn what program counter is, how it works and many more. So, let's start –

Before going to the direct topic, let us discuss some primary terminologies such as:

### ***CPU (Central Processing Unit)***

The full name of the CPU is Central Processing Unit. It is also known as a processor. CPU is the brain of the computer system.

It is an electronic microchip that processes the data and converts it into useful information based on the instructions given and controls all the functions of the computer system.

### **Instructions:-**

Computer instructions are a group of machine language instructions and they are executed by a specific computer processor. Whatever instruction we give to the computer, it will work on its basis.

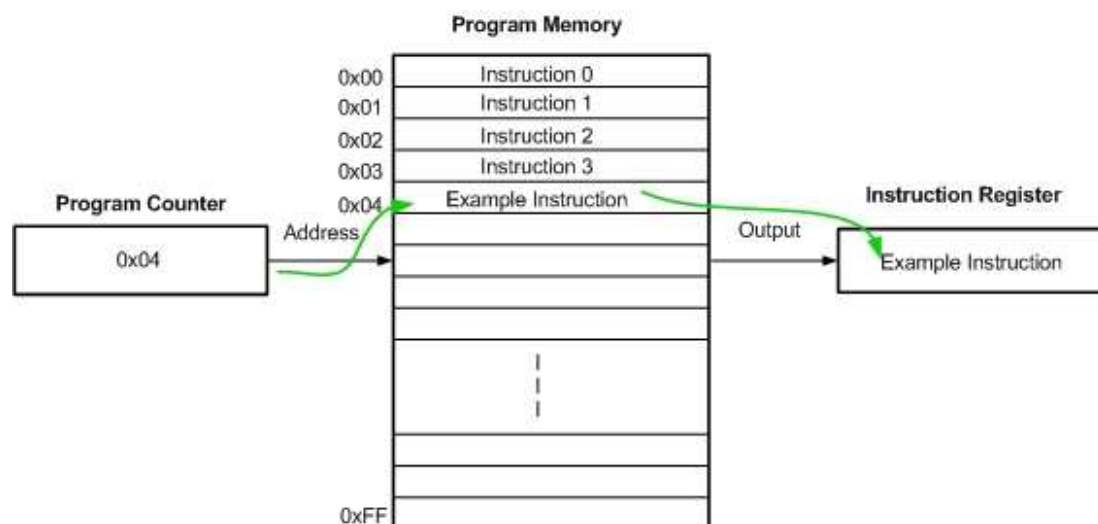
### **Memory:-**

Computer memory is a device that is used to store data and information. In other words, "Computer memory is an important part of the computer in which data is stored, without memory the computer does not work." .Memory includes RAM and ROM.

### ***What is Program Counter?***

There is a register in a PC (program counter) processor that contains the address of the next instruction to be executed from memory.

It is a 16-bit register and is also called instruction counter, instruction pointer, and instruction address register (IAR). PC (program counter) is a digital counter that is needed to execute tasks quickly and track the current execution point.



All instructions and data in memory have a specific address. As each instruction is processed, the Program Counter (PC) updates to point to the next instruction. When a byte (machine code) is fetched, the PC increases by one to prepare for the next instruction. If the computer is reset or restarted, the PC resets to zero.

For example, if the PC shows 8000H, it means the processor will fetch the instruction at that address. After fetching the byte at 8000H, the PC automatically increments by one, getting ready to fetch the next instruction or opcode.

### **FAQs on Program Counter**

#### **Q.1: Can the program counter be modified?**

**Answer:**

Yes, the program counter can be modified by certain instructions or events during program execution. For example, branching instructions can change the program counter to redirect the flow of execution to a different part of the program.

---

#### **Q.2: What happens when the program counter is modified?**

**Answer:**

When the program counter is modified, the CPU will fetch the instruction from the new address specified by the modified program counter. This allows for non-sequential execution and enables features like loops, conditionals, and function calls in programming.

---

#### **Q.3: Is the program counter the same as a memory address?**

**Answer:**

No, the program counter points to the next instruction to be executed, while a memory address refers to a specific location in memory where data or instructions are stored.

---

#### **Q.4: Can the program counter go backwards?**

**Answer:**

Typically, the program counter moves forward sequentially. However, certain instructions, like jumps or loops, can cause the program counter to move backward or to a different position in memory.

---

#### **Q.5: What happens if the program counter points to an invalid address?**

**Answer:**

If the program counter points to an invalid address, it can lead to a program crash or an error.

The CPU may attempt to fetch an instruction from an invalid memory location, resulting in undefined behavior or an exception.

---

## Bus Structures

- **Bus Definition:** A bus is a subsystem that transfers data between computer components, either within a computer or between two computers. It connects peripheral devices simultaneously.
- **Types of Bus Structures:**
  - **Single Bus Structure:**
    - Simple setup with one bus.
    - All units are connected to the same bus.
    - Cost-effective but lower performance.
  - **Multiple Bus Structure:**
    - Consists of interconnected buses.
    - Each bus can connect to others as foreign buses.
    - Better performance than single bus structures.
- **Key Features:**
  - A bus cannot span multiple cells.
  - Each cell can have more than one bus.
  - No messaging engine in a single bus structure.
  - Communication lines are used for data, addresses, and control information.
- **Example:** Data transfer from processor to printer, often using buffer registers to hold content during transfer.

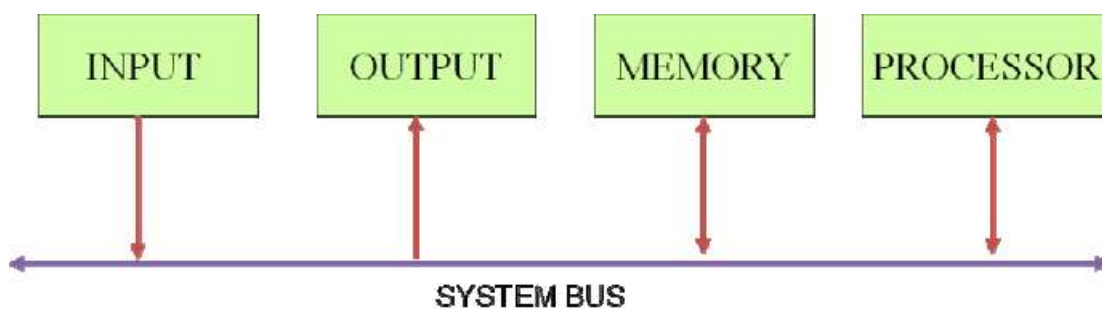


Figure 5: Single bus structure

**Buffer Registers:** Buffer registers temporarily hold data during transfer.

**Example:** Used in printing.

## Types of Buses

### *Data Bus*

- The **data bus** is the most common type, used to transfer data between different components of a computer.
- The number of lines in the data bus affects the speed of data transfer. It typically has 8, 16, 32, or 64 lines.
- A 64-line data bus can transfer 64 bits of data at one time.
- The data bus lines are bi-directional, meaning:
  - The CPU can read data from memory.
  - The CPU can write data to memory.

### ***Address Bus***

- Components are connected via buses, with each assigned a unique ID called the **address**.
- When a component wants to communicate with another, it uses the address bus to specify the target address.
- The address bus is **unidirectional**, carrying information in one direction—from the microprocessor to the main memory.

### ***Control Bus***

- The **control bus** transmits commands and control signals between components.
- For example, if the CPU wants to read data from main memory, it uses the control bus.
- Control signals include:
  - **Timing Information:** Specifies how long a device can use the data and address bus.
  - **Command Signal:** Indicates the type of operation to perform (e.g., writing data).
- After completing a command, the memory sends an acknowledgment signal back to the CPU, which then continues with other actions.

## **Software**

If a user wants to enter and run an application program, they need **system software**. System software is a collection of programs that perform functions such as:

- Receiving and interpreting user commands
- Entering, editing, and storing application programs as files on secondary storage devices
- Running standard application programs like word processors, spreadsheets, games, etc.

The **operating system** is a key component of system software, helping users interact with the underlying hardware through programs.

## **Types of Software**

### **System Software**

System software helps run computer hardware and the system itself. It includes:

- Device drivers



- Operating systems
- Servers
- Utilities
- Windowing systems
- Compilers
- Debuggers
- Interpreters
- Linkers

The purpose of system software is to simplify the tasks of application programmers by managing complex details of the computer, including communication devices, printers, displays, and keyboards. It also partitions the computer's resources, like memory and processor time, in a safe and stable way. Examples include Windows XP, Linux, and macOS.

### **Application Software**

Application software enables end users to accomplish specific tasks that are not directly related to computer development. Typical applications include:

- Word processors
- Spreadsheets
- Computer Games
- Business Software
- Chemistry & physics software
- Databases
- Medical software
- Military Software
- Tele Communications
- Image editing software
- Decision Making Software

Application software exists for a wide variety of topics and has had a significant impact in many areas.

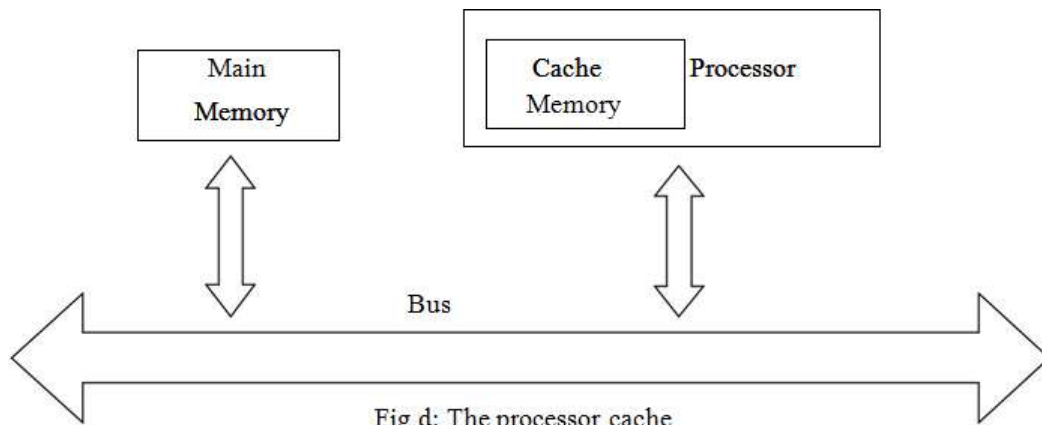
### **Performance**

The most important measure of a computer's performance is how quickly it can execute programs. The speed of program execution is influenced by the design of its hardware. For optimal performance, the compiler, machine instruction set, and hardware should be designed together.

**Elapsed Time:** The total time required to execute a program is called elapsed time and measures the performance of the entire computer system. This time is affected by the speed of the processor, disk, and printer.

**Processor Time:** The time needed to execute an instruction is called processor time.

Both elapsed time and processor time depend on the hardware involved in executing individual machine instructions. This includes the processor and memory, which are typically connected by a bus.



The relevant parts of the first figure are repeated in the second figure, which includes cache memory as part of the processor unit.

### Program Execution Flow

At the start of execution, all program instructions and required data are stored in the main memory. As execution proceeds, instructions are fetched one by one over the bus into the processor, with a copy placed in the cache. If the same instruction or data is needed again, it is read directly from the cache.

The processor and cache memory can be built on a single integrated circuit (IC) chip. The internal speed of instruction processing on the chip is very high and much faster than fetching instructions and data from the main memory. Programs execute faster when the movement of instructions and data between main memory and the processor is minimized, which is achieved by using cache.

**Example:** If a set of instructions is executed repeatedly, as in a program loop, having those instructions in the cache allows for quick fetching during repeated use. The same applies to frequently used data.

### Data Representation

Registers are made up of flip-flops, which are two-state devices that can store only 1s and 0s.

## Number Base Conversion Methods

There are various methods to convert numbers from one base to another. We will demonstrate the following:

- Decimal to Other Base
- Other Base to Decimal
- Other Base to Non-Decimal
- Shortcut Method: Binary to Octal
- Shortcut Method: Octal to Binary
- Shortcut Method: Binary to Hexadecimal
- Shortcut Method: Hexadecimal to Binary

### 1. Decimal to Other Base

#### *Decimal to Binary*

1. **Divide** the decimal number by 2.
2. **Record the remainder** (0 or 1).
3. **Repeat** with the quotient until it is 0.
4. **Read the remainders** from bottom to top.

#### *Example:-*

1. Divide by 2:  
 $25 \div 2 = 12$  remainder **1**  
 $12 \div 2 = 6$  remainder **0**  
 $6 \div 2 = 3$  remainder **0**  
 $3 \div 2 = 1$  remainder **1**  
 $1 \div 2 = 0$  remainder **1**
2. Reading from bottom to top: **11001**  
**Binary: 25 = 11001**

#### *Decimal to Octal*

1. **Divide** the decimal number by 8.
2. **Record the remainder.**
3. **Repeat** until the quotient is 0.
4. **Read the remainders** from bottom to top.

#### *Example:-*

1. Divide by 8:  
 $25 \div 8 = 3$  remainder **1**  
 $3 \div 8 = 0$  remainder **3**
2. Reading from bottom to top: **31**  
**Octal: 25 = 31**

### ***Decimal to Hexadecimal***

1. **Divide** the decimal number by 16.
2. **Record the remainder** (0-9, A-F).
3. **Repeat** until the quotient is 0.
4. **Read the remainders** from bottom to top.

#### ***Example:-***

1. Divide by 16:  
 $25 \div 16 = 1$  remainder **9**  
 $1 \div 16 = 0$  remainder **1**
2. Reading from bottom to top: **19**  
**Hexadecimal: 25 = 19**

## **2. Other Base to Decimal**

### ***Binary to Decimal***

1. **Multiply** each bit by  $2^n$  (where n is the position from the right, starting at 0).
2. **Sum the results.**

#### ***Example:-***

$$\begin{aligned}
 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
 &= 16 + 8 + 0 + 0 + 1 = 25 \\
 &= 25
 \end{aligned}$$

**Decimal: 11001 = 25**

### ***Octal to Decimal***

1. **Multiply** each digit by  $8^n$  (where n is the position from the right).
2. **Sum the results.**

***Example:-***

$$= 3 \times 8^1 + 1 \times 8^0$$

$$= 24 + 1 = 25$$

**Decimal: 31 = 25**

***Hexadecimal to Decimal***

1. **Multiply** each digit by  $16^n$  (where n is the position from the right).
2. **Sum** the results.

***Example:-***

$$1. \quad 1 \times 16^1 + 9 \times 16^0$$

$$2. \quad = 16 + 9 = 25$$

**Decimal: 19 = 25**

**3. Other Base to Non-Decimal**

***Binary to Octal***

1. **Group** binary digits in sets of three (from right).
2. **Convert** each group to its octal equivalent.

***Example:-***

1. Group binary in sets of 3 (from right): **011 001**

2. Convert:

○ **011 = 3**

○ **001 = 1**

**Octal: 11001 = 31**

***Octal to Binary***

1. **Convert** each octal digit to a 3-bit binary number.
2. **Combine** the binary numbers.

***Octal 31 to Binary***

1. Convert each digit to 3 bits:
  - a. **3 = 011**
  - b. **1 = 001**
2. Combine: **011001**

**Binary: 31 = 11001**

### ***Binary to Hexadecimal***

1. **Group** binary digits in sets of four (from right).
2. **Convert** each group to its hexadecimal equivalent.

### ***Binary 11001 to Hexadecimal***

1. Group binary in sets of 4 (from right): **0001 1001**
2. Convert:
  - **0001** = 1
  - **1001** = 9**Hexadecimal: 11001 = 19**

### ***Hexadecimal to Binary***

1. **Convert** each hexadecimal digit to a 4-bit binary number.
2. **Combine** the binary numbers.

### ***Hexadecimal 19 to Binary***

1. Convert each digit to 4 bits:
  - **1** = 0001
  - **9** = 1001
2. Combine: **00011001**  
**Binary: 19 = 11001**

### **Binary Coded Decimal (BCD) code**

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

## Binary System Complements

In the binary system (base 2), there are two main types of complements: **1's complement** and **2's complement**.

### 1's Complement:-

The **1's complement** of a binary number is obtained by flipping all bits in the number:

- Change every **1** to a **0**.
- Change every **0** to a **1**.

#### Example:

- Original binary number: **1010**
- 1's complement:
  - $1 \rightarrow 0$
  - $0 \rightarrow 1$
  - $1 \rightarrow 0$
  - $0 \rightarrow 1$
- **1's complement of 1010 is 0101.**

#### ***Key Points about 1's Complement:***

- **Addition/Subtraction:** To perform subtraction using 1's complement, add the 1's complement of the number to be subtracted and carry over any extra bit.
- **Range of Representation:** In a binary system using  $n$  bits, the 1's complement allows representation of numbers  $-(2^{n-1}-1)$  to  $+(2^{n-1}-1)$

### 2's Complement:-

The **2's complement** is another way of representing negative numbers in binary. It is found by taking the 1's complement of a number and then adding 1 to the least significant bit (LSB).

#### Example:

- Original binary number: **1010**
- Step 1: Find the 1's complement: **0101**
- Step 2: Add 1:

```
  0101
+ 0001
-----
  0110
```

- **2's complement of 1010 is 0110.**

### ***Key Points about 2's Complement:***

- **Unique Representation of Zero:** 2's complement has one representation for zero, unlike 1's complement, which has both +0 and -0.
- **Simplified Arithmetic:** Addition and subtraction can be performed directly using binary addition, making it simpler for digital circuits.
- **Range of Representation:** In a binary system using  $n$  bits, the 2's complement allows representation of numbers from  $-2^{n-1}$  to  $2^{n-1}-1$ .

## **Introduction to Programming Languages:**

**Computer Languages:** To write a program for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages.

1940's Machine level Languages

1950's Symbolic Languages

1960's High-Level Languages

**Machine Languages:** In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0's and 1's.

**Machine languages** are the most basic type of programming languages, directly understood by a computer's hardware. Here's a simple breakdown:

### ***1. What is Machine Language?***

- **Definition:** The lowest-level programming language, consisting of binary code (0s and 1s) that a computer's CPU can execute directly.

### ***2. Characteristics***

- **Binary Code:** Uses sequences of 0s and 1s to represent instructions.
- **Direct Execution:** Commands are executed directly by the computer's hardware.
- **No Abstraction:** Offers no abstraction from the computer's physical components.



### *3. Examples*

- **Specific to Each CPU:** Different CPUs (e.g., Intel, ARM) have different machine languages.
- **Instruction Set:** Each CPU has its own set of machine instructions, like moving data or performing arithmetic.

### *4. Advantages*

- **Fast Execution:** Code runs very quickly because it's directly executed by the hardware.
- **Efficient:** No overhead from translation or interpretation.

### *5. Disadvantages*

- **Complex:** Difficult for humans to read and write directly.
- **Error-Prone:** More prone to errors and harder to debug compared to higher-level languages.

## **Symbolic Languages:**

In early 1950's Admiral Grace Hopper, A mathematician and naval officer developed the concept of a special computer program that would convert programs into machine language.

### *1. What Are Symbolic Languages?*

- **Definition:** Languages that use symbols and expressions for programming.
- **Purpose:** Handle symbolic computation and logic.

### *2. Key Features*

- **Symbols and Expressions:** Use symbols (like variables) and expressions (like mathematical formulas) to represent and manipulate data.
- **High Abstraction:** Provide a high level of abstraction from machine hardware.
- **Dynamic Typing:** Often handle various data types flexibly.

### *3. Examples of Symbolic Languages*

- **LISP:**
  - **Use:** Symbolic computation and AI.
  - **Feature:** Uses lists as the main data structure.
  - **Example:** (defun square (x) (\* x x))

## High Level Languages:

High-level languages are programming languages that provide a high level of abstraction from the machine's hardware. They are designed to be easy for humans to read and write, making programming more accessible and productive. Here's a simple overview:

### 1. What Are High-Level Languages?

- **Definition:** Programming languages that allow you to write instructions in a more human-readable form, abstracting away complex machine code details.

### 2. Characteristics

- **Abstraction:** Abstract away the hardware specifics, focusing on logic and functionality.
- **Readable Syntax:** Use syntax that is easier for humans to understand, similar to natural language or mathematical notation.
- **Portability:** Code can often run on different types of hardware with minimal changes.

### 3. Examples

- **Python:** Easy to read and write, widely used for web development, data analysis, and automation.
  - **Example Code:** `print("Hello, World!")`
- **Java:** Object-oriented language used for building platform-independent applications.

- **Example Code:**

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

- **C++:** Builds on C with object-oriented features, used for system/software development and performance-critical applications.

- **Example Code:**

```
#include <iostream>  
using namespace std;  
  
void main() {  
    cout << "Hello, World!" << endl;  
}
```

- **JavaScript:** Primarily used for web development to add interactivity to websites.

- **Example Code:** `console.log("Hello, World!");`

#### *4. Advantages*

- **Ease of Use:** Simplifies complex tasks with more intuitive syntax.
- **Productivity:** Speeds up development time with built-in libraries and frameworks.
- **Maintenance:** Easier to maintain and update code due to its readability.

#### *5. Disadvantages*

- **Performance:** May be slower compared to lower-level languages due to abstraction.
- **Less Control:** Less direct control over hardware and memory compared to low-level languages.

### **Translator Programs:**

**Translator programs** are software tools that convert code written in one programming language into another language or format. They enable developers to write code in high-level languages and run it on different systems or environments. Here's a simple overview:

#### *1. What Are Translator Programs?*

- **Definition:** Software that translates code from one form to another to make it executable or compatible with different systems.

#### *2. Types of Translator Programs*

##### **1. Compilers:**

- **Purpose:** Convert high-level source code into machine code (binary) that the computer's CPU can execute.
- **Process:** Translates the entire program before execution.
- **Examples:** GCC (for C/C++), javac (for Java).
- **Pros:** Typically results in faster execution since the translation is done ahead of time.
- **Cons:** Compilation can take time, and errors are detected during the compile phase.

##### **2. Interpreters:**

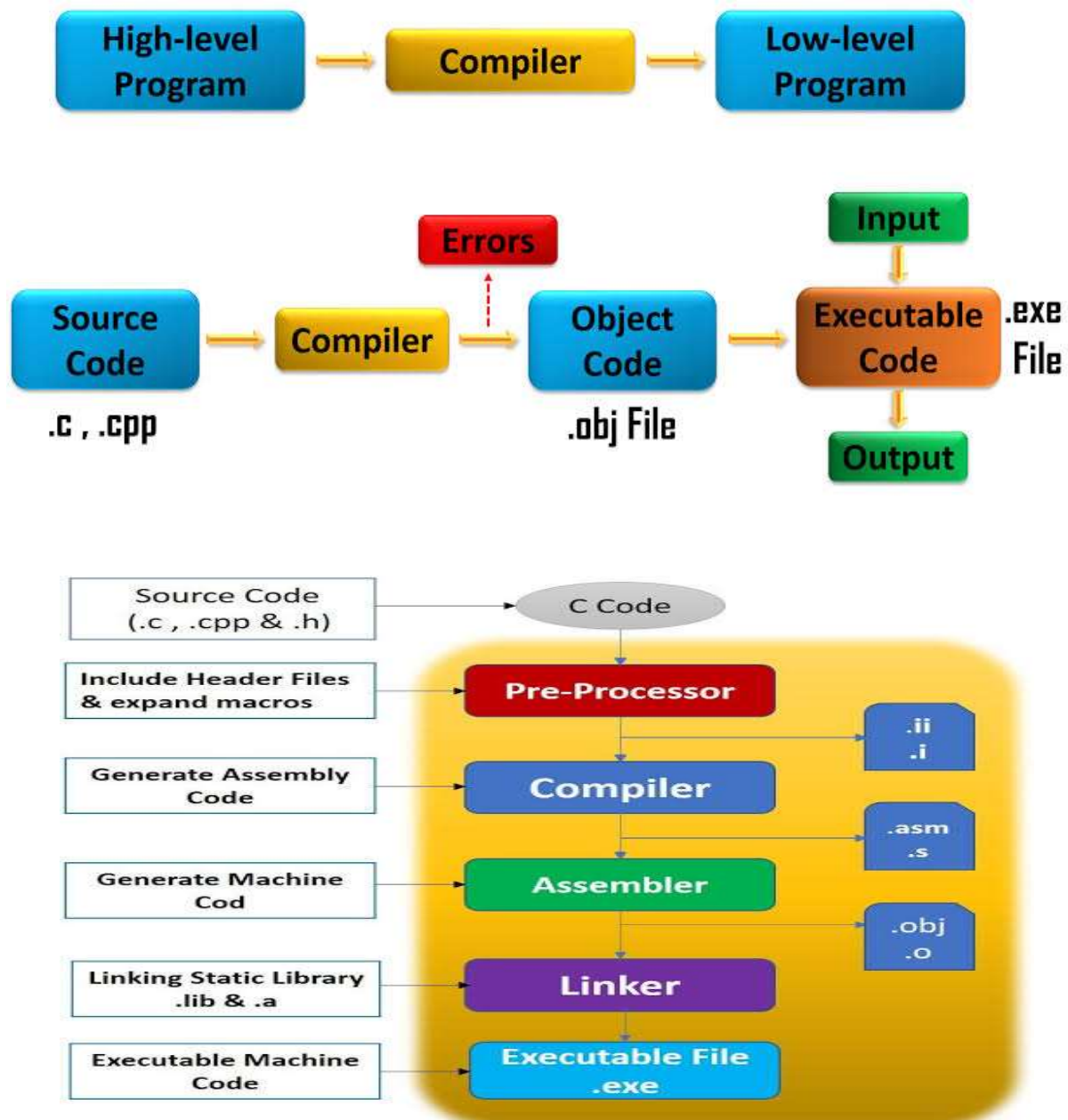
- **Purpose:** Execute high-level source code directly without converting it into machine code first.

- **Process:** Translates and runs code line-by-line or statement-by-statement.
- **Examples:** Python interpreter, Ruby interpreter.
- **Pros:** Easier to debug and test code in smaller increments.
- **Cons:** Slower execution since code is translated at runtime.

### 3. Assemblers:

- **Purpose:** Convert assembly language code into machine code.
- **Process:** Translates assembly instructions (which are human-readable) into binary code.
- **Examples:** NASM, MASM.
- **Pros:** Makes assembly language programs executable by converting them to machine code.
- **Cons:** Assembly language is often complex and hardware-specific.

Compiler	Interpreter
A compiler translate the compiler source program in a single line	An interpreter translate the source program line by line
It is faster	It is slower
It consumes less time	It consumes more time them compiler
It is more efficient	It is less efficient
Compilers are larger in size	Interpreters are smaller than compilers



Computer languages can be categorized in various ways depending on their level of abstraction, use cases, and programming paradigms. Here's a broad overview:

#### 1. By Level of Abstraction:

- **Low-Level Languages:**

- **Machine Language:** The most fundamental level, consisting of binary code that the computer's CPU directly executes.

- **Assembly Language:** A human-readable representation of machine language instructions, using mnemonics. Requires an assembler to translate into machine code.
- **High-Level Languages:**
  - **Procedural Languages:** Focus on a sequence of steps or procedures (e.g., C, Pascal).
  - **Object-Oriented Languages:** Emphasize objects and classes, encapsulating data and behavior (e.g., Java, C++, Python).
  - **Functional Languages:** Focus on mathematical functions and immutability (e.g., Haskell, Lisp).
  - **Scripting Languages:** Often used for automating tasks, with easier syntax and dynamic typing (e.g., Python, JavaScript, Ruby).

## 2. By Paradigm:

- **Imperative Languages:** Specify commands for the computer to perform (e.g., C, Fortran).
- **Declarative Languages:** Describe what the program should accomplish without specifying how (e.g., SQL, HTML).
- **Logic Languages:** Based on formal logic, with rules and facts used to derive conclusions (e.g., Prolog).
- **Event-Driven Languages:** Designed around the concept of events and handlers (e.g., JavaScript in web development).

## 3. By Use Case:

- **Systems Programming Languages:** Used for developing system software (e.g., C, Rust).
- **Application Programming Languages:** Used for creating applications (e.g., Java, C#).
- **Web Programming Languages:** Specifically designed for web development (e.g., JavaScript, PHP).
- **Database Languages:** Specialized for managing and querying databases (e.g., SQL).
- **Domain-Specific Languages (DSLs):** Tailored for specific problem domains (e.g., MATLAB for mathematical computing, VHDL for hardware description).

## 4. By Typing Discipline:

- **Static Typing:** Type checking is done at compile-time (e.g., Java, C++).
- **Dynamic Typing:** Type checking is done at runtime (e.g., Python, JavaScript).
- **Strong Typing:** Enforces strict type constraints (e.g., Haskell).
- **Weak Typing:** Allows more flexibility with types (e.g., JavaScript).

Each language has its strengths and weaknesses, and the choice of language often depends on the specific needs of a project or task.

**Programming languages are tools used to give instructions to a computer. They allow you to write software, from simple scripts to complex systems. Here's a basic overview:**

### ***1. What is a Programming Language?***

- **Definition:** A formal set of rules and symbols used to create software.
- **Components:**
  - **Syntax:** Rules for structuring code (e.g., how to write a command).
  - **Semantics:** Meaning of the code constructs (e.g., what a command does).
  - **Pragmatics:** How the language is used in practice (e.g., coding best practices).

### ***2. Types of Programming Languages***

- **High-Level Languages:** Abstract away hardware details. Example: Python.
- **Low-Level Languages:** Close to machine code. Example: Assembly.
- **Compiled Languages:** Converted to machine code before running. Example: C++.
- **Interpreted Languages:** Executed line-by-line by an interpreter. Example: JavaScript.
- **Hybrid Languages:** Can be both compiled and interpreted. Example: Java.

### ***3. Programming Paradigms***

- **Imperative Programming:** Describes a sequence of commands. Example: C.
- **Object-Oriented Programming (OOP):** Uses objects to organize code. Example: Java.
- **Functional Programming:** Focuses on functions and immutability. Example: Haskell.
- **Logic Programming:** Uses logic and rules. Example: Prolog.
- **Declarative Programming:** Specifies what should be done, not how. Example: SQL.

### ***4. Evolution of Programming Languages***

- **Early Languages:** Assembly and Fortran.
- **Structured Programming:** Introduced with languages like C.
- **Object-Oriented Programming:** Developed with languages like C++ and Java.
- **Scripting and Web Development:** Emerged with languages like JavaScript and Python.
- **Modern Trends:** Focus on concurrency, safety, and performance (e.g., Rust, Go).

### ***5. Choosing a Programming Language***

- **Project Requirements:** Pick based on the task (e.g., JavaScript for web development).
- **Performance Needs:** Choose based on efficiency (e.g., C for performance).
- **Development Speed:** Consider ease of use (e.g., Python for rapid development).
- **Community and Ecosystem:** Look for support and libraries (e.g., Python's extensive libraries).

## **Programming Timeline:**

Timeline with key programming milestones, highlighting the year, the programmer or developer, and the significant achievement:

### **1800s**

- **1801: Joseph Marie Jacquard** - Invents the Jacquard loom, using punched cards.

### **1830s**

- **1837: Charles Babbage** - Designs the Analytical Engine. **Ada Lovelace** writes the first algorithm for this machine.

### **1940s**

- **1941: Konrad Zuse** - Develops the Z3, the world's first programmable digital computer.
- **1943-1944: British Codebreakers** - Use the Colossus computers for decryption.
- **1945: John Presper Eckert and John Mauchly** - Complete the ENIAC, one of the first general-purpose electronic digital computers.

### **1950s**

- **1951: Remington Rand** - Releases the UNIVAC I, the first commercially available computer.
- **1957: IBM Team (led by John Backus)** - Releases Fortran, one of the first high-level programming languages.
- **1958: John McCarthy** - Develops Lisp, an early language for artificial intelligence.

### **1960s**

- **1964: IBM** - Introduces COBOL, a high-level language for business data processing.
- **1965: Edsger W. Dijkstra** - Popularizes structured programming principles.
- **1969: Dennis Ritchie and Ken Thompson** - Develop Unix and the C programming language.

### **1970s**

- **1972: Dennis Ritchie** - Releases the C programming language.
- **1975: Bill Gates and Paul Allen** - Found Microsoft and develop BASIC for early personal computers.
- **1978: Brian Kernighan and Dennis Ritchie** - Publish "The C Programming Language."



## 1980s

- **1983: Richard Stallman** - Launches the GNU Project to create a free Unix-like operating system.
- **1983: Ada Lovelace** - The Ada programming language is adopted by the U.S. Department of Defense.

## 1990s

- **1991: Guido van Rossum** - Creates Python.
- **1994: James Gosling** - Releases Java, a language designed for cross-platform compatibility.

## 2000s

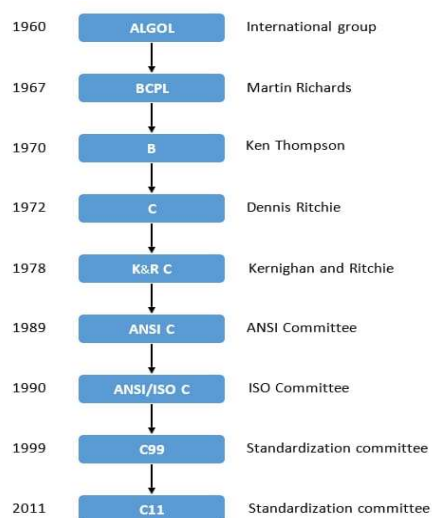
- **2000: Agile Alliance** - Promotes the Agile Manifesto, emphasizing iterative development and collaboration.
- **2001: Microsoft Team (led by Anders Hejlsberg)** - Releases C#, a modern language for the .NET framework.

## 2010s

- **2010: Various Developers** - The rise of JavaScript frameworks like AngularJS and React changes web development practices.
- **2014: Apple Inc. (led by Chris Lattner)** - Introduces Swift as a modern alternative to Objective-C.

## 2020s

- **2020: Various Contributors** - The rise of AI, machine learning, and new programming paradigms continues to shape the field.



**What is the general structure of a 'C' program and explain with example?**

Structure of C Programming:-

**1) \* Documentation section \*/**

**2) /\* Link section \*/**

**3) /\* Definition section \*/**

**4) /\* Global declaration section \*/**

**main() {**

**4.1) Declaration part**

**4.2) Executable part (statements)**

**}**

**5) /\* Sub-program section \*/**

### **1. Documentation Section :**

This section consists of a set of comment lines giving the name of the program, and other details which the programmer would like to use later.

Ex:- /\*Addition of two numbers \*/

### **2. Link section:**

Link section provides instructions to the compiler to link functions from the system library.

Ex:- #include<stdio.h>

#include<conio.h>

### **3. Definition section:**

Definition section defines all symbolic constants. Ex:- #  
define A 10.

#### 4. Global declaration section:

Some of the variables that are used in more than one function throughout the program are called global variables and declared outside of all the functions. This section declares all the user-defined functions.

#### main() function section:

Every C program must have one main ( ) function section. This contains two parts.

1. **Declaration part:** This part declares all the variables used in the executable part. Ex:- int a,b;
2. **Executable part:** This part contains at least one statement. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. All the statements in the declaration and executable parts end with a semicolon (;).

**5. Sub program section:** This section contains all the user-defined functions, that are called in the main ( ) function. User- defined functions are generally placed immediately after the main() function, although they may appear in any order.

Ex:



## **Basics of a Computer Program-Algorithms:**

A computer program is a sequence of instructions that a computer executes to perform a specific task or solve a problem. To create effective programs, understanding the basics of algorithms is crucial. Here's an overview:

### **1. What is an Algorithm?**

An algorithm is a step-by-step procedure or a set of rules designed to perform a task or solve a problem. Algorithms are the foundation of programming and guide how a computer processes data.

### **2. Characteristics of Algorithms**

- **Clear and Unambiguous:** Each step must be precisely defined and understandable.
- **Finite:** Algorithms must have a clear stopping point, meaning they must terminate after a finite number of steps.
- **Input and Output:** An algorithm should take zero or more inputs and produce at least one output.
- **Effective:** Each step of the algorithm should be simple enough to be performed exactly and in a finite amount of time.

### **3. Basic Components of Algorithms**

1. **Initialization:** Setting up the initial values or conditions.
2. **Input:** Taking in data that will be processed by the algorithm.
3. **Processing:** The core part of the algorithm where operations are performed on the input data.
4. **Output:** Producing the result or the solution.
5. **Termination:** The end of the algorithm, where it stops after completing its task.

### **4. Representing Algorithms**

- **Natural Language:** Describing the algorithm using everyday language (e.g., "To make tea, boil water, add tea leaves, let steep, pour into cup").
- **Pseudocode:** Writing the algorithm in a structured but informal way that is easy to understand (e.g., START; BOIL water; ADD tea leaves; STEEP tea; POUR into cup; END).
- **Flowcharts:** Using diagrams with symbols to represent the flow of steps (e.g., rectangles for processes, diamonds for decisions).

### **5. Common Types of Algorithms**

- **Sorting Algorithms:** Organize data in a specific order (e.g., Bubble Sort, Merge Sort, Quick Sort).

- **Search Algorithms:** Find specific data within a structure (e.g., Binary Search, Linear Search).
- **Recursive Algorithms:** Solve problems by solving smaller instances of the same problem (e.g., Factorial calculation, Fibonacci sequence).

## 6. Example of an Algorithm

**Task:** Find the largest number in a list.

**Algorithm:**

1. **Initialize:** Set largest to the first element of the list.
2. **Input:** Read the list of numbers.
3. **Process:**
  - For each number in the list (starting from the second element):
    - If the number is greater than largest, update largest to this number.
4. **Output:** Return the value of largest.
5. **Termination:** End when all numbers have been processed.

**Pseudocode:**

```
START
largest ← list[0]
FOR each number in list
  IF number > largest THEN
    largest ← number
  ENDIF
END FOR
RETURN largest
END
```

## 7. Analyzing Algorithms

Algorithms are often analyzed for efficiency, including:

- **Time Complexity:** How the execution time of the algorithm grows with the size of the input (e.g.,  $O(n)$ ,  $O(\log n)$ ).
- **Space Complexity:** How the amount of memory used grows with the size of the input.

## 8. Practical Considerations

- **Correctness:** The algorithm must produce the correct result for all possible inputs.
- **Efficiency:** The algorithm should be efficient in terms of both time and space.
- **Readability:** The algorithm should be easy to understand and maintain.

Understanding and designing algorithms effectively is essential for programming because they

provide the blueprint for solving problems with code. Once you have a well-defined algorithm, implementing it in a programming language becomes much more straightforward.

### **Examples on Algorithm:**

Algorithm was developed by an Arab mathematician. It is chalked out step-by-step approach to solve a given problem. It is represented in an English like language and has some mathematical symbols like  $\rightarrow$ ,  $>$ ,  $<$ ,  $=$  etc. To solve a given problem or to write a program you approach towards solution of the problem in a systematic, disciplined, non-adhoc, step-by-step way is called Algorithmic approach. Algorithm is a penned strategy(to write) to find a solution.

Example: Algorithm/pseudo code to add two numbers

Step 1: Start

Step 2: Read the two numbers in to a, b

Step 3:  $c=a+b$

Step 4: write/print c


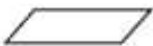


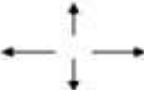




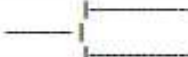
Step 5: Stop.

**Flowcharts:** A flowchart is a diagrammatic representation of an algorithm

The diagrammatic representation of an algorithm is called a **flowchart**.

A flowchart visually depicts the steps and decisions involved in an algorithm or process using various symbols and shapes. This graphical format helps in understanding, analyzing, and communicating the logic of the algorithm or workflow in a clear and organized manner.

Here are some of the commonly used symbols in flowcharts:

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing task.
		Preprocessor
		Comments

A flowchart is a visual representation of the sequence of steps for solving a problem . A flowchart is a set of symbols that indicate various operations in the program. For every process , there is a corresponding symbol in the flowchart . Once an algorithm is written , its pictorial representation can be done using flowchart symbols.

In other words, a pictorial representation of a textual algorithm is done using a flowchart. A flowchart gives a pictorial representation of an algorithm.

The first flowchart is made by John Von Neumann in 1945.

It is a symbolic diagram of operations sequence, dataflow, control flow and processing

Logic in information processing. The symbols used are simple and easy to learn.

It is a very helpful tool for programmers and beginners.

**Purpose of a Flowchart :** Provides communication.

- ❖ Provides an overview.
- ❖ Shows all elements and their relationships.
- ❖ Quick method of showing program flow.
- ❖ Checks program logic.
- ❖ Facilitates coding.
- ❖ Provides program revision.
- ❖ Provides program documentation.

**Advantages of a Flowchart :** Flowchart is an important aid in the development of an algorithm itself.

- ❖ Easier to understand than a program itself.
- ❖ Independent of any particular programming language.
- ❖ Proper documentation
- ❖ Proper debugging.
- ❖ Easy and clear presentation.

**Limitations of a Flowchart :** Complex logic.

- ❖ Drawing is time consuming.
- ❖ Difficult to draw and remember.
- ❖ Technical detail

**Write an algorithm and flow chart for swapping two numbers**

Ans:

To Swap two integer numbers:

**Algorithm :** using third variable

Step 1 : Start

Step 2 : Input num1 , num2

Step 3 : temp = num1

Step 4 : num1 = num2

Step 5 : num2 = temp

Step 6 : Output num1 , num2

Step 7 : Stop

**Algorithm :** without using third variable

Step 1 : Start

Step 2 : Input num1 , num2

Step 3 : calculate num1 = num1 + num2

Step 4 : calculate num2 = num1 - num2

Step 5 : calculate num1 = num1 - num2



Step 6 : Output num1 , num2

Step 7 : Stop

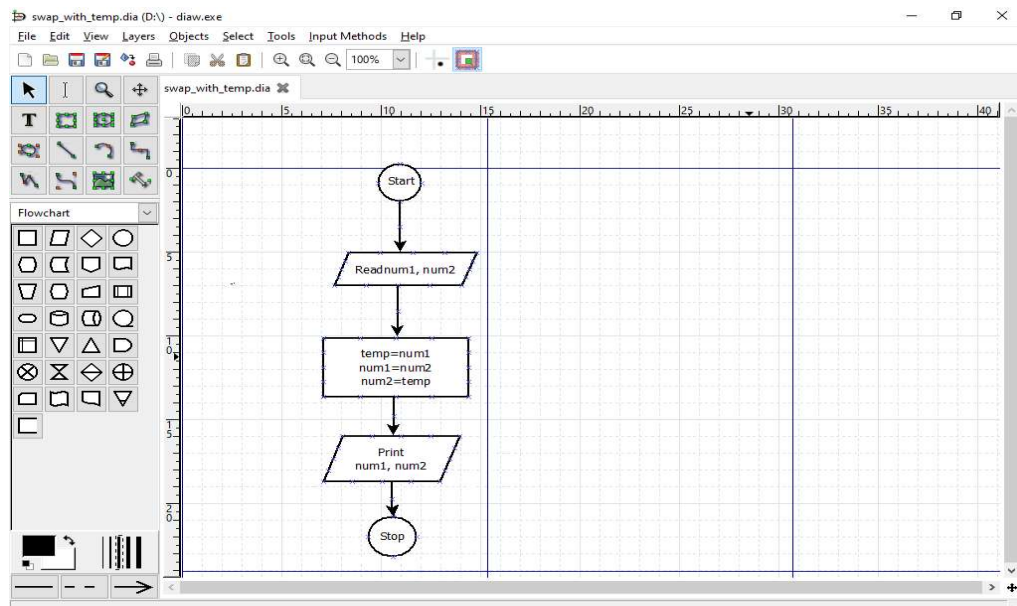


Fig 1: With using third variable

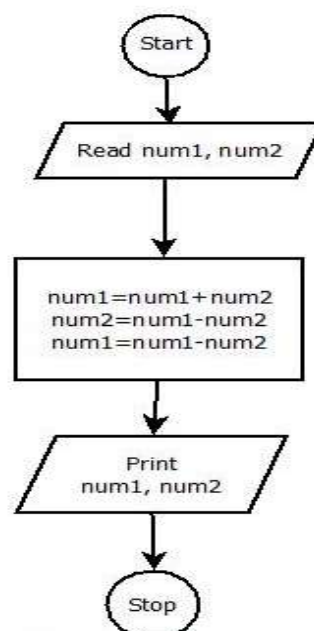


Fig 2: Without using third variable

Fig 2: Without using third variable

## Pseudocode:-

Pseudocode is a way to describe the logic of a program in a simple, human-readable format that doesn't follow any specific programming language syntax. It's often used to outline the steps in an algorithm before you write actual code. Here's a basic guide on how to write pseudocode:

### Basic Structure of Pseudocode

1. **Start and End:** Indicate where the algorithm begins and ends.

```
START  
...  
END
```

2. **Variables:** Declare variables and assign values.

```
SET variable_name TO value
```

3. **Input/Output:** Describe how to take input from the user and display output.

```
READ input_variable  
PRINT output_variable
```

4. **Control Structures:** Use simple keywords for decision-making and looping.

#### ❖ If Statements:

```
IF condition THEN  
... (actions)  
ELSE  
... (alternative actions)  
ENDIF
```

#### ❖ Loops:

```
while:  
• WHILE condition DO  
... (actions)  
ENDWHILE
```

#### ❖ For:

```
FOR i FROM start TO end DO  
... (actions)  
ENDFOR
```

❖ **Functions/Procedures:** Define reusable blocks of code.

```
FUNCTION function_name(parameters)
... (actions)
RETURN result
ENDFUNCTION
```

### Example Pseudocode

Let's say you want to write pseudocode for finding the maximum of two numbers:

```
➡ START
➡ READ number1
➡ READ number2

➡ IF number1 > number2 THEN
➡ SET maximum TO number1
➡ ELSE
➡ SET maximum TO number2
➡ ENDIF

➡ PRINT maximum
➡ END
```

In this example:

- START and END define the boundaries of the algorithm.
- READ is used to take user input.
- IF is used to compare the numbers.
- PRINT is used to display the result.

## **Introduction to Compilation and Execution:-**

### **1. Writing the Code**

The process begins with writing the source code in C. This is done using a text editor or an Integrated Development Environment (IDE). The source code typically has a .c extension.

### **2. Preprocessing**

The first step in the compilation process is preprocessing. The preprocessor handles directives that start with #, such as #include, #define, and #if. It performs tasks like:

- **File Inclusion:** The #include directive is used to include the contents of a file, typically header files, into the source code.
- **Macro Substitution:** #define is used to define macros which are replaced with their

values.

- **Conditional Compilation:** The `#if`, `#ifdef`, and `#endif` directives are used for compiling code conditionally.

The result is a translation unit which is a modified version of the original source file.

### 3. Compilation

The next step is compilation, where the preprocessed code is translated into assembly code. This step involves:

- **Syntax and Semantic Analysis:** The compiler checks for syntax errors and ensures that the code follows the rules of the C language.
- **Intermediate Code Generation:** The compiler generates an intermediate representation of the source code.
- **Optimization:** The compiler may optimize the intermediate code to improve performance and reduce resource usage.

The output of the compilation step is usually an assembly code or an object code (typically with a `.o` or `.obj` extension).

### 4. Assembly

In this step, the assembler converts the assembly code into machine code. This machine code is in binary format and is specific to the architecture of the target machine. The output is an object file, which contains machine code and additional information needed for linking.

### 5. Linking

The final step is linking, where the linker combines object files and resolves references between them. It performs tasks such as:

- **Combining Object Files:** It combines multiple object files into a single executable.
- **Resolving External References:** It links functions and variables that are used across different object files.
- **Library Linking:** It incorporates code from libraries (e.g., standard libraries) that the program uses.

The result of this step is an executable file (typically with a `.exe` extension on Windows or no extension on Unix-like systems).

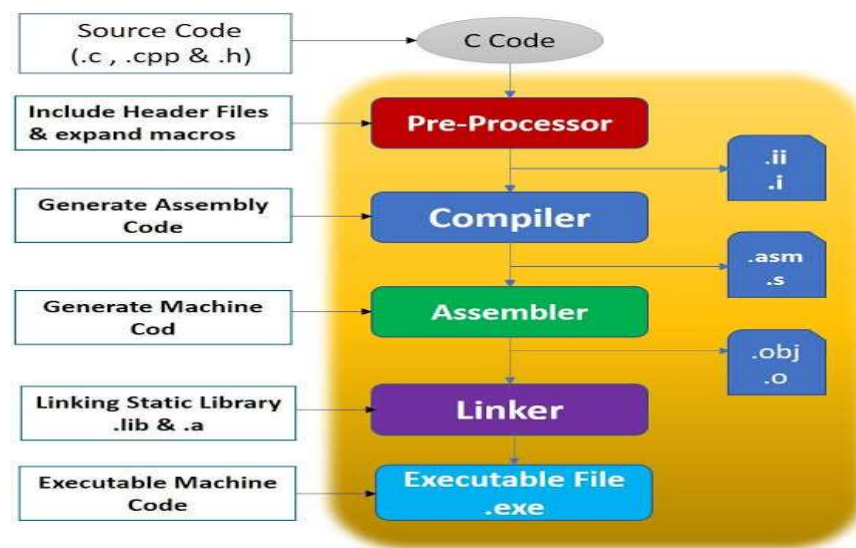
### 6. Execution

The executable file is now ready to be run. When executed, the operating system loads the program into memory and begins executing it. During execution, the program interacts with the

operating system, performs computations, and may produce output or accept input.

### Summary:

1. **Write** source code (.c file).
2. **Preprocess** to handle directives and macros.
3. **Compile** the preprocessed code into object code.
4. **Assemble** the object code into machine code.
5. **Link** the object files and libraries to create the executable.
6. **Execute** the program.



**Character Set:-** A character set is a collection of characters that can be represented in the programming language. The C language primarily uses the ASCII (American Standard Code for Information Interchange) character set, which includes:

- 1) **Letters:** Both uppercase (A-Z) and lowercase (a-z) alphabets.
- 2) **Digits:** The numerals from 0 to 9.
- 3) **Special Characters:** These include symbols like +, -, \*, /, =, <, >, &, |, !, ^, ~, %, #, \, ;, :, ', " and more.
- 4) **Whitespace Characters:** These include spaces, tabs, newlines, and form feeds.

### C-Tokens:

In C programming, "tokens" are the smallest units of a program that the compiler recognizes as meaningful. Tokens are the building blocks of a C program and are classified into several categories. Here's a breakdown of the different types of tokens in C:

## 1. Keywords

Keywords are reserved words that have special meaning in C and cannot be used as identifiers. They define the syntax and structure of the C language. Examples include:

- int
- float
- if
- else
- while
- return

## 2. Identifiers

Identifiers are names given to various program elements, such as variables, functions, arrays, etc. They must start with a letter (uppercase or lowercase) or an underscore, followed by letters, digits, or underscores. Examples include:

- main
- count
- totalSum

## 3. Constants

Constants represent fixed values that do not change during the execution of the program. They can be of different types:

- **Integer Constants:** E.g., 100, -42
- **Floating-point Constants:** E.g., 3.14, -0.001
- **Character Constants:** E.g., 'a', '%'

## 4. String Literals

String literals are sequences of characters enclosed in double quotes. They represent text values. Examples include:

- "Hello, World!"
- "C Programming"

## 5. Operators

Operators perform operations on variables and values. They are divided into several categories:

- **Arithmetic Operators:** +, -, \*, /, %
- **Relational Operators:** ==, !=, >, <, >=, <=

- **Logical Operators:** &&, ||, !
- **Bitwise Operators:** &, |, ^, ~, <<, >>
- **Assignment Operators:** =, +=, -=, \*=, /=, %=
- **Increment and Decrement Operators:** ++, --
- **Conditional Operator:** ? :

## 6. Punctuation or Separators

Punctuation tokens are symbols that help in organizing the structure of the code. They include:

- **Semicolon (;):** Used to terminate statements.
- **Comma (,):** Used to separate items in lists or function arguments.
- **Period (.):** Used to access a member of a structure.
- **Arrow (->):** Used to access a member of a structure through a pointer.
- **Parentheses ((, )):** Used to group expressions and parameters.
- **Braces ({, }):** Used to define blocks of code.
- **Brackets ([, ]):** Used to declare and access array elements.

## 7. Comments

Comments are not tokens in the strictest sense as they are ignored by the compiler, but they are used to annotate the code. Comments can be:

- **Single-line Comments:** Begin with //

```
// This is a single-line comment
```

- **Multi-line Comments:** Enclosed between /\* and \*/

```
/*
This is a multi-line comment
*/
```

## Example

Here's a simple example demonstrating various tokens in a C program:

```
#include <stdio.h> // '#include' is a preprocessor directive

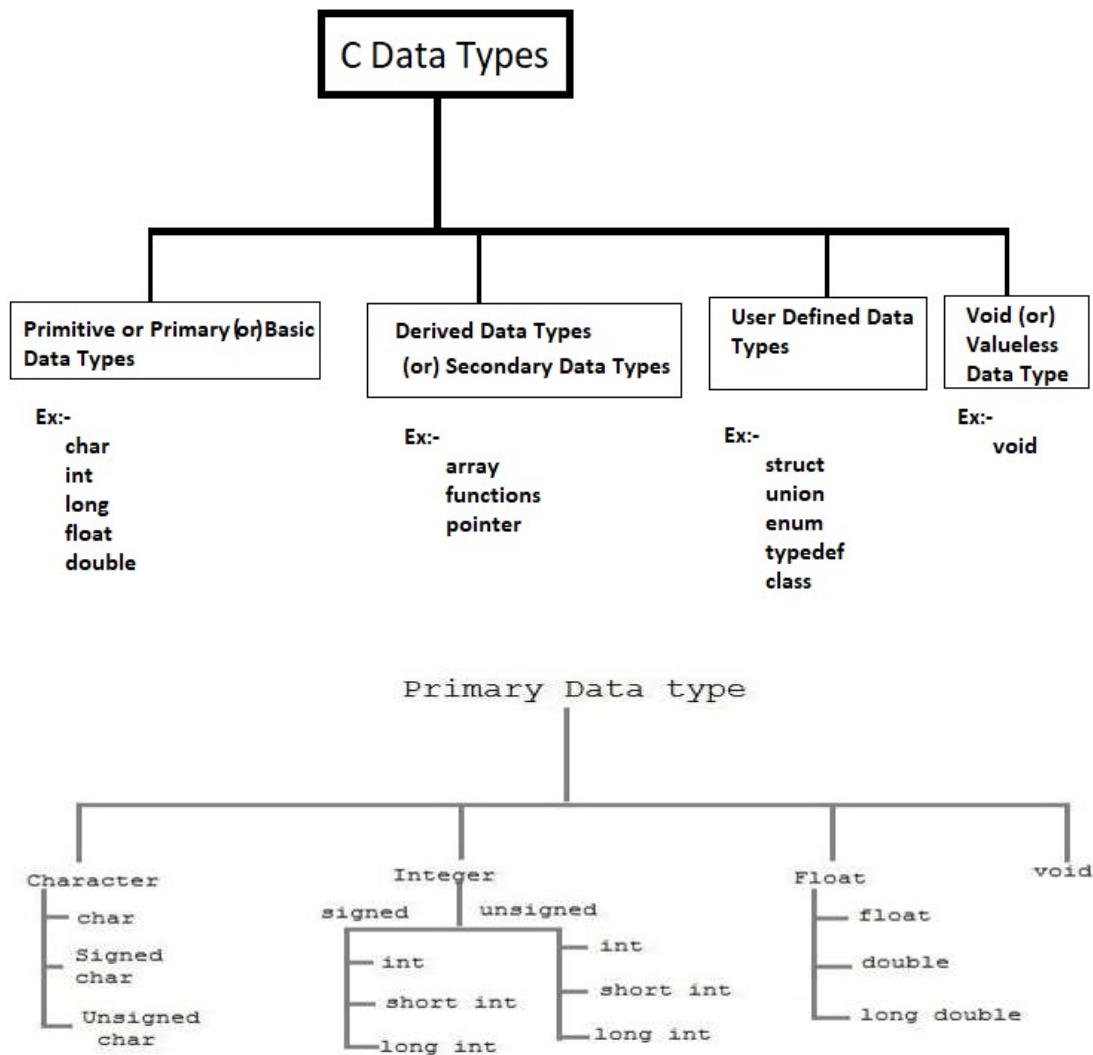
int main() { // 'int' and 'main' are keywords and identifiers
int x = 10; // 'int', 'x', '=' and '10' are tokens (identifier, keyword, operator, constant)
float y = 20.5; // 'float', 'y', '=' and '20.5' are tokens
printf("Sum: %f\n", x + y); // 'printf' is a function, '"Sum: %f\n"' is a string literal, '+' is an operator
return 0; // 'return', '0' are tokens
}
```

In this example:

- `#include` is a preprocessor directive, not a token in the same sense but part of the preprocessing phase.
- `int`, `main`, `float`, `return` are keywords.
- `x`, `y` are identifiers.
- `10`, `20.5`, `0` are constants.
- `=`, `+`, `;` are operators and punctuation.
- `"Sum: %f\n"` is a string literal.

Understanding these tokens and their roles is fundamental in learning C programming, as they form the basic syntax and structure of the language.

## Data Types:





In C programming, data types define the kind of data that can be stored and manipulated within a program. They determine the size and layout of the data in memory.

## 1. Primitive Data Types

Primitive data types are the basic types provided by the language. They are built into the language and are the simplest form of data types.

Data type	Size(bytes)	Range	Format string
Char	1	128 to 127	%c
Unsigned char	1	0 to 255	%c
Short or int	2	-32,768 to 32,767	%i or %d
Unsigned int	2	0 to 65535	%u
Long	4	-2147483648 to 2147483647	%ld
Unsigned long	4	0 to 4294967295	%lu
Float	4	3.4 e-38 to 3.4 e+38	%f or %g
Double	8	1.7 e-308 to 1.7 e+308	%lf
Long Double	10	3.4 e-4932 to 1.1 e+4932	%Lf

### ***Integer Types***

- int: Represents whole numbers. Sizes can vary but are usually 4 bytes (32 bits).
- short: A smaller version of int, typically 2 bytes (16 bits).
- long: A larger version of int, often 4 or 8 bytes (32 or 64 bits).
- long long: Extended size integer, typically 8 bytes (64 bits).

### ***Unsigned Integer Types***

- unsigned int: Non-negative integers.
- unsigned short: Non-negative integers, typically 2 bytes.
- unsigned long: Non-negative integers, typically 4 or 8 bytes.
- unsigned long long: Non-negative integers, typically 8 bytes.

### ***Character Types***

- char: Represents a single character, typically 1 byte (8 bits).
- signed char: Can represent both positive and negative values.
- unsigned char: Can only represent non-negative values.

### ***Floating-Point Types***

- float: Single precision, typically 4 bytes (32 bits).
- double: Double precision, typically 8 bytes (64 bits).
- long double: Extended precision, size can vary but often 12 or 16 bytes (96 or 128 bits).

## 2. Derived Data Types

Derived data types are built from primitive data types. They are used to create more complex data structures.

### *Arrays*

- **Definition:** Collection of elements of the same type.
- **Example:**

```
int numbers[10]; // Array of 10 integers
```

### *Pointers*

- **Definition:** Variables that store memory addresses.
- **Example:**

```
int *ptr; // Pointer to an integer
```

### *Functions*

- **Definition:** Special type that represents a block of code designed to perform a specific task.
- **Example:**

```
int add(int a, int b) {  
    return a + b;  
}
```

## 3. User-Defined Data Types

User-defined data types are created by the programmer to suit specific needs. They offer more complex data management.

### *Structures (struct)*

- **Definition:** Allows grouping different types of data into a single unit.
- **Example:**

```
struct Person {  
    char name[50];
```

```
int age;  
};
```

### ***Unions (union)***

- **Definition:** Allows storing different data types in the same memory location, but only one at a time.
- **Example:**

```
union Data {  
    int i;  
    float f;  
    char str[20];  
};
```

### ***Enumerations (enum)***

- **Definition:** Defines a set of named integer constants.
- **Example:**

```
enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

## **4. Void Data Type**

The void type represents the absence of type. It is used in various contexts:

### ***Void Functions***

- **Definition:** Functions that do not return a value.
- **Example:**

```
void printMessage() {  
    printf("Hello, World!\n");  
}
```

### ***Void Pointers***

- **Definition:** Pointers that can point to any data type, but cannot be dereferenced directly.
- **Example:**

```
void *ptr;  
int value = 10;  
ptr = &value;
```

## Summary

- **Primitive Data Types:** Basic types like int, char, float, and double.
- **Derived Data Types:** Includes arrays, pointers, and functions.
- **User-Defined Data Types:** Includes structures, unions, and enumerations.
- **Void Data Type:** Represents the absence of type or functions that do not return a value.

## Variable:

### 1. What is a Variable?

A variable in C is a named storage location in memory that holds a value of a specific data type. Each variable has a name, a type, and a value.

(or)

A variable is name, which holds a single value of specific data type. It is varying or updating a variable value and it has address represented with ‘&’ symbol. Based on variable data type it has a memory size and range. This is called a variable.

### **2. Declaration of Variables (or) Declaring Variables:**

These variables declared in declaration part. And then assigning in execution part.

Must assign before printing these variables otherwise get an error.

#### **Syntax:**

```
data_type variable_name;
```

#### **Example-1:**

```
int age;  
float salary;  
char grade;
```

#### **Example-2:**

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

```
void main() {
```

```
int x; // Declaration without initialization
```

```
printf("%d\n", x); // Undefined behavior: x is uninitialized

}
```

### 3. Initializing Variables

You can initialize a variable at the time of declaration by assigning it a value with ‘=’ assignment operator, these variables are used in execution part.

#### Syntax:

```
data_type variable_name = value;
```

#### Example:

```
int age = 25;
float salary = 50000.50;
char grade = 'A';
```

### 4. Variable Types

Variables can be of different types, which are categorized into:

#### 4.1. Basic Data Types

- int: Integer values. Can be short, int, long, or long long.
- float: Single-precision floating-point numbers.
- double: Double-precision floating-point numbers.
- char: Single characters.

#### 4.2. Modifiers

Modifiers can be used to alter the size and range of the data types:

- signed: Default for int, can represent negative and positive values.
- unsigned: Represents only non-negative values.
- short: Reduces size (e.g., short int).
- long: Increases size (e.g., long int).

#### Example:

```
unsigned int positiveNumber;
long long bigNumber;
```

## 5. Variable Naming Rules

Variable names in C must follow these rules:

- **Must start with a letter (a-z, A-Z) or an underscore (\_).**
- **Can contain letters, digits (0-9), and underscores.**
- **Cannot start with a digit.**
- **Cannot be a reserved keyword (e.g., int, return).**

### Example of Valid Names:

```
int student_age;  
float salary_amount;  
char first_name;
```

### Example of Invalid Names:

```
int 2ndPlace; // Starts with a digit  
float salary-amount; // Contains a hyphen  
int return; // Keyword
```

## 6. Variable Scope

Variables have different scopes, determining where they can be accessed:

- **Local Variables:** Declared inside a function or block and accessible only within that function or block.
- **Global Variables:** Declared outside all functions and accessible throughout the program.
- **Static Variables:** Retain their value between function calls. Declared with the static keyword.

### Example:

```
#include <stdio.h>  
  
int globalVar = 10; // Global variable  
  
void function() {  
    int localVar = 5; // Local variable  
    static int staticVar = 1; // Static variable  
    printf("Local: %d, Static: %d\n", localVar, staticVar);  
    staticVar++;  
}  
  
void main() {  
    printf("Global: %d\n", globalVar);  
    function();  
}
```

```
function(); // Static variable retains its value
}
```

## 7. Variable Types and Storage Classes

In addition to the data types, C has different storage classes that define the lifetime and visibility of variables:

- **auto:** Default storage class for local variables (automatic duration).
- **register:** Suggests to store the variable in a CPU register for faster access (not guaranteed).
- **static:** Retains the variable's value between function calls.
- **extern:** Refers to variables declared outside the current file or function, useful for accessing global variables from different files.

### Example:

```
#include <stdio.h>

int globalVar = 20; // global variable

void function() {
    static int staticVar = 0; // static variable
    auto int autoVar = 1; // auto is default, not necessary to use explicitly
    register int regVar = 2; // register storage class
    extern int globalVar; // extern variable
    printf("Static: %d, Auto: %d, Register: %d, Global: %d\n", staticVar, autoVar, regVar, globalVar);
    staticVar++;
}

void main() {
    function();
    function();
}
```

## 8. Examples of Variable Usage

Here's a simple example that demonstrates variable declaration, initialization, and usage:

```
#include <stdio.h>

int main() {
    int a = 10;    // Declare and initialize an integer variable
    float b = 20.5; // Declare and initialize a float variable
    char c = 'A';  // Declare and initialize a char variable

    printf("Integer: %d\n", a);
```

```
printf("Float: %.2f\n", b);
printf("Character: %c\n", c);

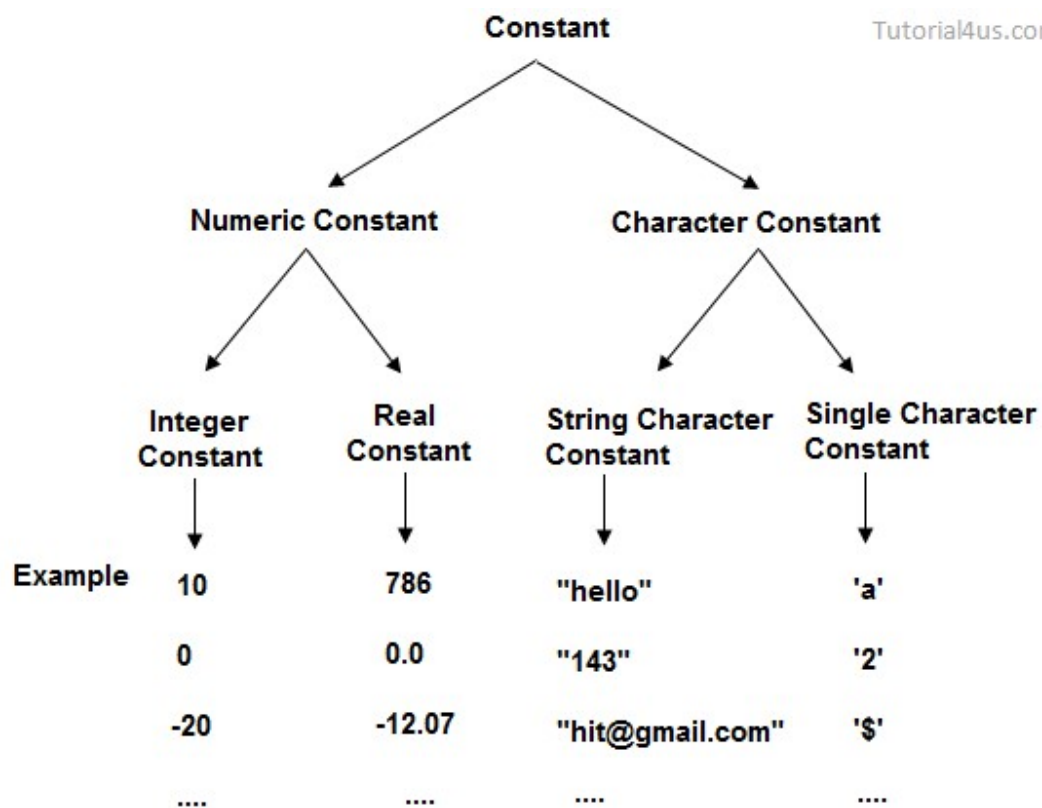
return 0;
}
```

## Summary

- **Declaring Variables:** Specify type and name.
- **Initializing Variables:** Assign an initial value at declaration.
- **Variable Types:** Includes basic types, modifiers, and storage classes.
- **Naming Rules:** Follow specific naming conventions.
- **Scope:** Local, global, and static.
- **Storage Classes:** auto, register, static, extern.

## Types of Constant in C:

In general constant can be used to represent as fixed values in a C program. Constants are classified into following types.





If any single character (alphabet or numeric or special symbol) is enclosed between single cotes " known as single character constant.

If set of characters are enclosed between double cotes "" known as string character constant.

## Backslash Characters (or) Escape Sequences:

Escape sequences in C are used to represent characters that are not easily typed on a keyboard or that have special meanings. They are introduced by a backslash (\) followed by a specific character. Here are some commonly used escape sequences:

Escape Sequences	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null
\a	Bell
\b	Backspace
\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Constants are values that cannot be modified during the execution of a program.

There are several ways to define constants in C:

### 1. Symbolic constants (or) Using #define Preprocessor Directive:

The #define directive defines a macro that can be used as a constant throughout your code.

symbolic constants are named constants whose values are set at compile time and cannot be changed during program execution.

```
#define PI 3.14159
#define MAX_SIZE 100
```

In this example, PI and MAX\_SIZE are replaced by their values wherever they appear in the code. Note that #define does not allocate memory for the constants; it simply replaces text.

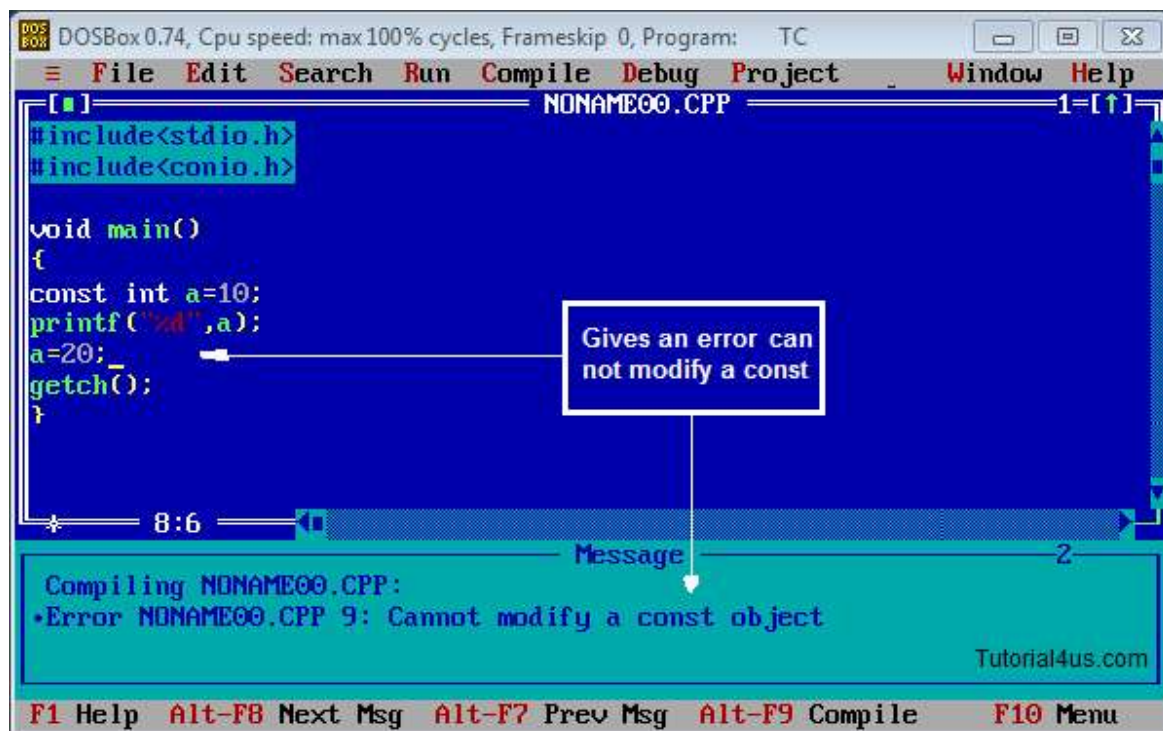
## 2. Declare constant: Using const Keyword

The const keyword defines a constant variable. Unlike #define, const variables are typed and can be debugged like other variables.

```
const double pi = 3.14159;  
const int maxSize = 100;
```

In this case, pi and maxSize are constants whose values cannot be modified after initialization. They also occupy memory space.

### Example:



## 3. Using enum for Integer Constants

The enum keyword can be used to define a set of named integer constants. This is often used for defining constants that are related in a meaningful way.

```
enum {  
    RED,  
    GREEN,
```

```
BLUE  
};
```

Here, RED, GREEN, and BLUE are constants with values 0, 1, and 2, respectively. You can also explicitly assign values to the constants:

```
enum {  
    RED = 1,  
    GREEN = 2,  
    BLUE = 4  
};
```

basic input and output operations are commonly handled using the standard library functions provided in the `stdio.h` header file. Here's a quick overview of how to perform these operations:

#### **Input Function:**

`scanf()` is a predefined function in "`stdio.h`" header file. It can be used to read the input value from the keyword.

#### **Syntax:**

```
scanf("format specifiers",&variable1,&variable2,...);
```

```
int scanf(const char *format, ...);
```

#### **Example:**

```
#include <stdio.h>  
  
void main() {  
    int num;  
    printf("Enter an integer: ");  
    scanf("%d", &num);  
    printf("You entered: %d\n", num);  
}
```

In this example:

- `printf` is used to prompt the user.
- `scanf` reads an integer from the user and stores it in the variable `num`.

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%lf	Long double

### Output Function:

The printf() is a predefined function in "stdio.h" header file, by using this function, we can print the data or user defined message on console or monitor

### Syntax:

```
1) printf("user defined message");
2) printf("Format specifiers",value1,value2,..);
3) printf("Format specifiers",variable1,variable2,..);
```

```
int printf(const char *format, ...);
```

### Example:

```
#include <stdio.h>
void main() {
    int age = 25;
    printf("Your age is %d years.\n", age);
}
```

In this example:

- printf formats and prints the value of age.

### Format Specifiers

Format specifiers are placeholders in the format string for variables. Here are some common format specifiers:

- %d or %i for integers
- %f for floating-point numbers
- %c for characters
- %s for strings
- %x for hexadecimal representation

## Examples

### Reading and Printing Multiple Values:

```
#include <stdio.h>

void main() {
    int age;
    float height;
    char initial;

    printf("Enter your age, height (in meters), and initial: ");
    scanf("%d %f %c", &age, &height, &initial);
    printf("Age: %d\n", age);
    printf("Height: %.2f meters\n", height);
    printf("Initial: %c\n", initial);
}
```

In this example:

- scanf reads an integer, a float, and a character.
- printf displays these values.

### Reading a String:

```
#include <stdio.h>

void main() {
    char name[50];

    printf("Enter your name: ");
    scanf("%49s", name); // Limit input to avoid buffer overflow

    printf("Hello, %s!\n", name);
}
```

In this example:

- %49s limits the input to 49 characters to prevent overflow, as name can hold up to 50 characters (including the null terminator).

## **Operator:**

Operator is a symbol, it is used between two operands.

Ex:  $a+b$

→  $a$  and  $b$  are operands

→  $+$  is a operator.

## **Types of Operator:**

### **1. Arithmetic Operators**

#### **Common Arithmetic Operators**

Operator	Description	Example
+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	$a / b$
%	Modulus (Remainder)	$a \% b$

#### **Example:**

```
int a=10,b=2;
```

- $+$  (Addition):  $a + b \Rightarrow 12$
- $-$  (Subtraction):  $a - b \Rightarrow 8$
- $*$  (Multiplication):  $a * b \Rightarrow 20$
- $/$  (Division):  $a / b \Rightarrow 5$
- $\%$  (Modulus):  $a \% b \Rightarrow 0$

<https://onlinegdb.com/nrugJy2Gd>

```
main.c
1 // including header files
2 #include<stdio.h>
3 // void main() function
4 void main(){
5     // 1. Declaration part
6     // Initialization of the variables
7     int a,b;
8     // 2. Execution part
9     // Arithmetic operators (+,-,*,/,%)
10
11     printf("Enter A & B:");
12     scanf("%d%d",&a,&b);
13
14     printf("A=%d,B=%d",a,b);
15     printf("\nA+B=%d",(a+b));
16     printf("\nA-B=%d",(a-b));
17     printf("\nA*B=%d",(a*b));
18     printf("\nA/B=%d",(a/b));
19     printf("\nA%%B=%d",(a%b));
20 }
21
```

```
Enter A & B:1
2
A=1,B=2
A+B=3
A-B=-1
A*B=2
A/B=0
A%B=1
```

## 2) Assignment operator

Assignment operators are used to assign values to variables. The most basic assignment operator is the single equals sign (=), but there are several other compound assignment operators that combine an arithmetic operation with assignment.

- a) Simple Assignment operator (=):- Assigns the value on the right to the variable on the left.
- b) Compound Assignment operator (+=, -=, \*=, /=, %=)
- c) Chaining Assignment operator (Ex: a=b=c=100;)

### a) Simple Assignment operator(=):

→ Assign a value into variable

**Syntax:**

```
variable = value;
```

**Example:**

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

**2.b) Compound Assignment Operator:**

These operators perform an operation and then assign the result to a variable. Here's a list of the common compound assignment operators:

Operator	Description	Example
+=	Addition and assignment	$a += 3$ (equivalent to $a = a + 3$ )
-=	Subtraction and assignment	$a -= 3$ (equivalent to $a = a - 3$ )
*=	Multiplication and assignment	$a *= 3$ (equivalent to $a = a * 3$ )
/=	Division and assignment	$a /= 3$ (equivalent to $a = a / 3$ )
%=	Modulus and assignment	$a \% = 3$ (equivalent to $a = a \% 3$ )

- = (Simple assignment):  $a = b$
- += (Add and assign):  $a += b$  (equivalent to  $a = a + b$ )
- -= (Subtract and assign):  $a -= b$  (equivalent to  $a = a - b$ )
- \*= (Multiply and assign):  $a *= b$  (equivalent to  $a = a * b$ )
- /= (Divide and assign):  $a /= b$  (equivalent to  $a = a / b$ )
- %= (Modulus and assign):  $a \% = b$  (equivalent to  $a = a \% b$ )

**Example:**



```
main.c
8 #include <stdio.h>
9 void main(){
10 // declaration of the variable
11 int a,b,c;
12 // a) Simple assignment operator (=)
13 printf("\na) Simple assignment operator (=)\n");
14 a=1;
15 printf("A=%d\n",a);
16
17 // b) compound assignment operator
18 printf("\nb) compound assignment operator\n");
19 a+=1; // a=a+1;
20 printf("A+=1 :- %d\n",a);
21 a-=1; // a=a-1;
22 printf("A-=1 :- %d\n",a);
23 a*=1; // a=a*1;
24 printf("A*=1 :- %d\n",a);
25 a/=1; // a=a/1;
26 printf("A/=1 :- %d\n",a);
27 a%=1; // a=a%1;
28 printf("A Modulo =1 :- %d\n",a);
29
30 // c) chainging assignment operator
31 printf("\nc) chainging assignment operator\n");
32 a=b=c=100;
33 printf("A=%d\n",a);
34 printf("B=%d\n",b);
35 printf("C=%d",c);
36 }
```

input

```
a) Simple assignment operator (=)
A=1

b) compound assignment operator
A+=1 :- 2
A-=1 :- 1
A*=1 :- 1
A/=1 :- 1
A Modulo =1 :- 0

c) chainging assignment operator
A=100
B=100
C=100
```

### 3. Relational Operators:

Relational operator used between two values, I.e condition. Every condition must return either true or false.

Relational operators in C are used to compare two values or expressions. They evaluate the relationship between them and return a boolean result (either true or false). These operators are essential for making decisions in conditional statements, such as `if` and `while` loops.

#### List of Relational Operators

Operator	Description	Example	Returns
==	Equal to	a == b	True if a is equal to b
!=	Not equal to	a != b	True if a is not equal to b
>	Greater than	a > b	True if a is greater than b
<	Less than	a < b	True if a is less than b

Operator	Description	Example	Returns
>=	Greater than or equal to	a >= b	True if a is greater than or equal to b
<=	Less than or equal to	a <= b	True if a is less than or equal to b

### Syntax:

- 1) value1 <relation operator> value2
- 2) variable1 <relation operator> variable2

### Examples:

int a=1,b=2;

- == (Equal to): a == b result: 0
- != (Not equal to): a != b output: 1
- (Greater than): a > b output: 0
- < (Less than): a < b output: 1
- >= (Greater than or equal to): a >= b output 0
- <= (Less than or equal to): a <= b result: 1

### Example:

```
main.c
1 // Relational operator in c
2 // symbols: <, <=, >, >=, ==, !=
3 // It is used between the two values or variables
4 // for comparision.
5 #include<stdio.h>
6 // true=1, false=0
7 void main(){
8     // Initialization of the a=1 and b=2
9     int a=1,b=2;
10    // <
11    printf("A<B = %d\n",a<b); // 1
12    // <=
13    printf("A<=B = %d\n",a<=b); // 1
14    // >
15    printf("A>B = %d\n",a>b); // 0
16    // >=
17    printf("A>=B = %d\n",a>=b); // 0
18    // ==
19    printf("A==B = %d\n",a==b); // 0
20    // !=
21    printf("A!=B = %d\n",a!=b); // 1
22 }
23
```

input

```
A<B = 1
A<=B = 1
A>B = 0
A>=B = 0
A==B = 0
A!=B = 1
```

### 3.Logical Operators:

Logical operation are used between one or more conditions but **Logical NOT** applied only on single condition.

Used mixed of Logical AND, OR and NOT conditions in single statement.

→ Applied between the two conditions.

**3.a) Logical AND (&&):** ALL conditions are true then

the resultant of the logical AND is true otherwise false.

**Syntax:**

**1) Condition1 && Condition2**

→ All conditions must be true then the resultant of Logical AND (&&) is TRUE otherwise FALSE.

Let A and B are two conditions.

A   B   (A && B)

-----

T   T   T

T   F   F

F   T   F

F   F   F

-----

0= false

1= true

Applied three conditions.

Let A,B and C are two conditions.

A B C (A && B && C)

0 0 0 0

0 0 1 0

0 1 0 0

0 1 1 0

1 0 0 0

1 0 1 0

1 1 0 0

1 1 1 1

2) **Logical OR (||)**: Any one condition is true then the resultant of the

logical OR is true otherwise false.

**Syntax:**

**Condition1 || Condition2**

→ Any one conditions is TRUE then the resultant of Logical OR is TRUE otherwise FALSE.

Let A and B are two conditions

|| Trueth Table:-

-----

A B (A || B)

-----

T T T

T F T

F T T

F F F

-----

A B C (A || B || C)

0 0 0 0

0 0 1 1

0 1 0 1

0 1 1 1

1 0 0 1

1 0 1 1

1 1 0 1

1 1 1 1

### 3) Logical NOT (!) :

--> Applied only single condition.

Opposite results applied for one condition

**Syntax:**

**! Condition1**

Let A is condition

-----

A !A

-----

T F

F T

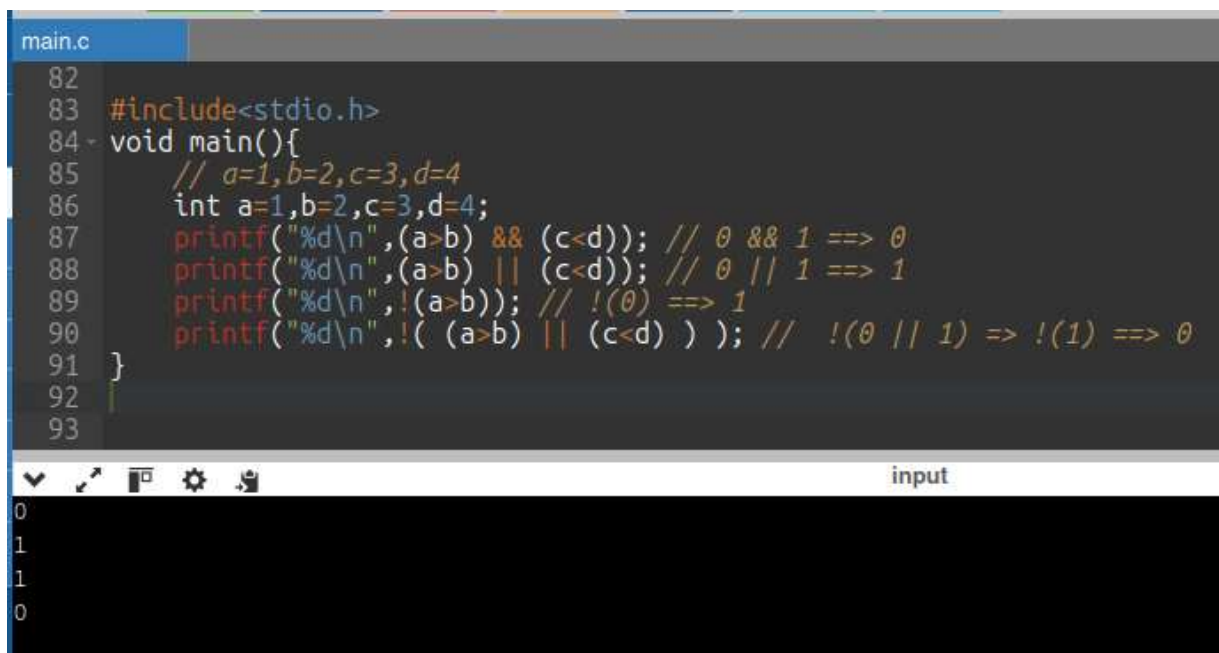
\*/

→ Opposite of condition result

Used to perform logical operations:

- ! (Logical NOT): !a

**Example:**



```
main.c
82
83 #include<stdio.h>
84 void main(){
85     // a=1,b=2,c=3,d=4
86     int a=1,b=2,c=3,d=4;
87     printf("%d\n",(a>b) && (c<d)); // 0 && 1 ==> 0
88     printf("%d\n",(a>b) || (c<d)); // 0 || 1 ==> 1
89     printf("%d\n",!(a>b)); // !(0) ==> 1
90     printf("%d\n",!( (a>b) || (c<d) ) ); // !(0 || 1) => !(1) ==> 0
91 }
92
93
```

input

0  
1  
1  
0

## 5. Increment and Decrement Operators

### Increment Operator (++)

- **Usage:** Increases the value of a variable by 1.

- **Types:**
  - **Postfix** ( $x++$ ): Increments  $x$ , but returns the original value before the increment.
  - **Prefix** ( $++x$ ): Increments  $x$ , and returns the new value after the increment.

### Decrement Operator ( $--$ )

- **Usage:** Decreases the value of a variable by 1.
- **Types:**
  - **Postfix** ( $x--$ ): Decrements  $x$ , but returns the original value before the decrement.
  - **Prefix** ( $--x$ ): Decrements  $x$ , and returns the new value after the decrement.

Used to increase or decrease the value of a variable:

- $++$  (Increment):  $a++$  or  $++a$

```
main.c
1  #include <stdio.h>
2
3  void main()
4  {
5      int i=5;
6      printf("i=%d\n",i); // i=5 #5
7      // preincrement operator (++i)
8      printf(++i=%d\n",++i); // ++i => i=i+1 => 5+1 => #6
9      printf("i=%d\n",i); // i=6
10     // postincrement operator (i++)
11     printf("i++=%d\n",i++); // i++ => i=i+1 => i=6+1=7 #6
12     printf("i=%d\n",i); // i=7 #7
13 }
14
```

input

```
i=5
++i=6
i=6
i++=7
i=7
```

- $--$  (Decrement):  $a--$  or  $--a$

```
main.c
1  #include <stdio.h>
2  void main()
3  {
4      int i=5;
5      printf("i=%d\n",i); // i=5
6      // Predecrement operator(--i)
7      printf("--i=%d\n",--i); // --i=> i=i-1 => 5-1 => 4
8      printf("i=%d\n",i); // i=4
9      // postdecrement operator (i--)
10     printf("i--=%d\n",i--); // i-- => i=i-1 =4-1=3 postpone to next line
11     printf("i=%d\n",i); // i=3
12 }
13
```

input

```
i=5
--i=4
i=4
i--=4
i=3
```

## 6. Conditional (Ternary) Operator (?:)

The conditional (ternary) operator in C is a concise way to perform conditional evaluations. It takes three operands and is often used as a shorthand for an `if-else` statement.

### Syntax:

- 1) (condition)? True-Part: False-Part;
- 2) variable = Expression1? Expression2: Expression3;

### **Example:**



```
main.c
5
6 Syntax:
7 1 ) condition? truepart:falsepart;
8 2 ) variable = condition? truepart:falsepart;
9
10 Example:
11 int a=1,b=2;
12 a<b?printf("true"):printf("false");
13 output:
14     true
15
16
17
18 *****
19 #include <stdio.h>
20
21 void main(){
22     int a=2,b=4,result;
23
24     a<b?printf("True"):printf("False");
25
26     result=a>b?a*b:a+b;
27     printf("\nResult is %d",result);
28 }
29
```

input

True  
Result is 6

## 7. Bitwise Operators

Operate on the binary representations of numbers:

- & (Bitwise AND):  $a \& b$
- | (Bitwise OR):  $a | b$
- ^ (Bitwise XOR):  $a \wedge b$
- ~ (Bitwise NOT):  $\sim a$
- << (Left shift):  $a \ll 2$
- >> (Right shift):  $a \gg 2$

Binary digit = 0 OR 1

1 bit format: 0 value = 0

2 bit format: 0 value = 00

3 bit format: 0 value = 000

4 bit format: 0 value = 0000

8 bit format 0 value = 00000000

8 4 2 1

0 = 0 0 0 0

1 = 0 0 0 1

2 = 0 0 1 0

3 = 0 0 1 1

4 = 0 1 0 0

5 = 0 1 0 1

6 = 0 1 1 0

7 = 0 1 1 1 =  $8*0+4*1+2*1+1*1=0+4+2+1=7$

8 4 2 1

6 = 0 1 1 0 =  $8*0+4*1+2*1+1*0=4+2=6$

3 = 0 0 1 1

| = Bitwise OR

-----

0 1 1 1 => 7

-----

/\*

8 4 2 1

6=0 1 1 0 =8\*0+4\*1+2\*1+1\*0=4+2=6

3=0 0 1 1

& = Bitwise AND

-----

0 0 1 0 => 2

-----

leftshift (<<):-

8 4 2 1

6=0 1 1 0

1 1 0 0=>12 digit moved leftside (<<)

Syntax:

(variable << MovingDigits)

Example:

1) 6<<1 = 12 Double value

2) 6<<2 = 24

8 4 2 1

6=0 1 1 0

0 0 1 1 => 1 digit moved leftside (<<)

1)  $6 \gg 1 = 3$  (half value)

2)  $6 \gg 2 = 1$

### EX-OR(Exclusive OR : ^)

-----

A	B	A ^ B
---	---	-------

-----

0	0	0
---	---	---

0	1	1
---	---	---

1	0	1
---	---	---

1	1	0
---	---	---

Example:

8421

6=> 0110

3=> 0011

^

-----

$0101 \Rightarrow 8*0+4*1+2*0+1*1 \Rightarrow 0+4+0+1 = 5$

-----

Tilt operator (~): It means ( +1 ) & opposite sign

1)  $a=6 \Rightarrow \sim a \Rightarrow a+1 = 6+1 = 7 = -7$

2)  $a=-6 \Rightarrow \sim a \Rightarrow -6+1=-5 = 5$

```
main.c
87
88 #include "stdio.h"
89 void main(){
90     int a=6,b=3;
91     printf("a&b => %d\n",a&b); // 2
92     printf("a|b => %d\n",a|b); // 7
93     printf("~ is tilt operator: ~b => %d\n",~b); // 3(+1) =4 ==> -4
94     printf("~-4 => %d\n",~-4); // -4(+1) =-3 ==> 3
95     printf("a<<1 => %d\n",a<<1); // 12
96     printf("a>>1 => %d\n",a>>1); // 3
97     printf("a^b => %d",a^b); // 5
98 }
99
100
101
```

input

```
a&b => 2
a|b => 7
~ is tilt operator: ~b => -4
~-4 => 3
a<<1 => 12
a>>1 => 3
a^b => 5
```

## 8. Comma Operator

Used to separate expressions in a single statement:

```
#include <stdio.h>

int main() {
    int a, b;

    // Chaining with comma operator
    a = (b = 1, b + 1);

    // Print values
    printf("a = %d\n", a); // Output: a = 2
    printf("b = %d\n", b); // Output: b = 1

    return 0;
}
```

## 9. Sizeof Operator

Determines the size, in bytes, of a data type or variable:

- sizeof: sizeof(int), sizeof(a)

```
main.c
1- /*
2- sizeof(): To print the size of the memory of data type
3-
4- Syntax:
5-
6-     int sizeof(Data-Type)
7-
8- */
9- #include <stdio.h>
10- #include <limits.h>
11- #include <float.h>
12-
13- void main(){
14-     // Integer data types
15-     printf("short int : %ld\n",sizeof(short int));
16-     printf("Size of signed int : %ld\n",sizeof(signed int));
17-     printf("Range of signed int %d to %d\n", INT_MIN, INT_MAX);
18-     printf("unsigned short int : %ld\n",sizeof(unsigned short int));
19-     printf("unsigned int : %ld\n",sizeof(unsigned int));
20-     printf("int : %ld\n",sizeof(int));
21-     printf("long : %ld\n",sizeof(long));
22-     // Char Data types
23-     printf("char : %ld\n",sizeof(char));
24-     // Floating point data types
25-     printf("float : %ld\n",sizeof(float));
26-     printf("double : %ld\n",sizeof(double));
27-     printf("long double : %ld\n",sizeof(long double));
28- }
```

input

```
short int : 2
Size of signed int : 4
Range of signed int -2147483648 to 2147483647
unsigned short int : 2
unsigned int : 4
int : 4
long : 8
char : 1
float : 4
double : 8
long double : 16
```

## 10. Pointer Operators

Operate on pointers and memory addresses:

- \* (Dereference): \*ptr (accesses the value at the address pointed to by ptr)
- & (Address-of): &var (gets the address of var)

/\*

### Pointer:

Pointer is a variable which holds the address of another variable.

Declaration of Pointer:

-----

Syntax:

<DataType> \*<pointerVariable>;

Example:

int \*ptr; // here \*ptr is a pointer variable.

--> pointer variable indicate with "\*".

Initialization Of pointer:

-----

Or Assigning a pointer:

-----

Syntax:

```
pointerVariable = &variableName;
```

Example:

```
int i=10; // "i" is a variable.
```

```
int *ptr; // ptr is pointer variable.
```

```
ptr=&i; // Assigning an address of variable to pointer.
```

```
*/
```

```
// Variable to Pointer Example
```

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

```
void main(){
```

```
// Initialization of variable
```

```
int n=10;
```

```
// declaration of pointer
```

```
int *ptr;
```

```
printf("\nn=%d",n);
```

```
printf("\nn=%X",&n);
```

```
// holds(Assigning) a address of another variable
```

```
ptr=&n;
```

```
printf("\nValue of *ptr=%d",*ptr);
```

```
printf("\nAddress of ptr=%X",ptr);
```

```
}
```

output:

```
n=10
```

```
n=543A7AAC
```



Value of \*ptr=10

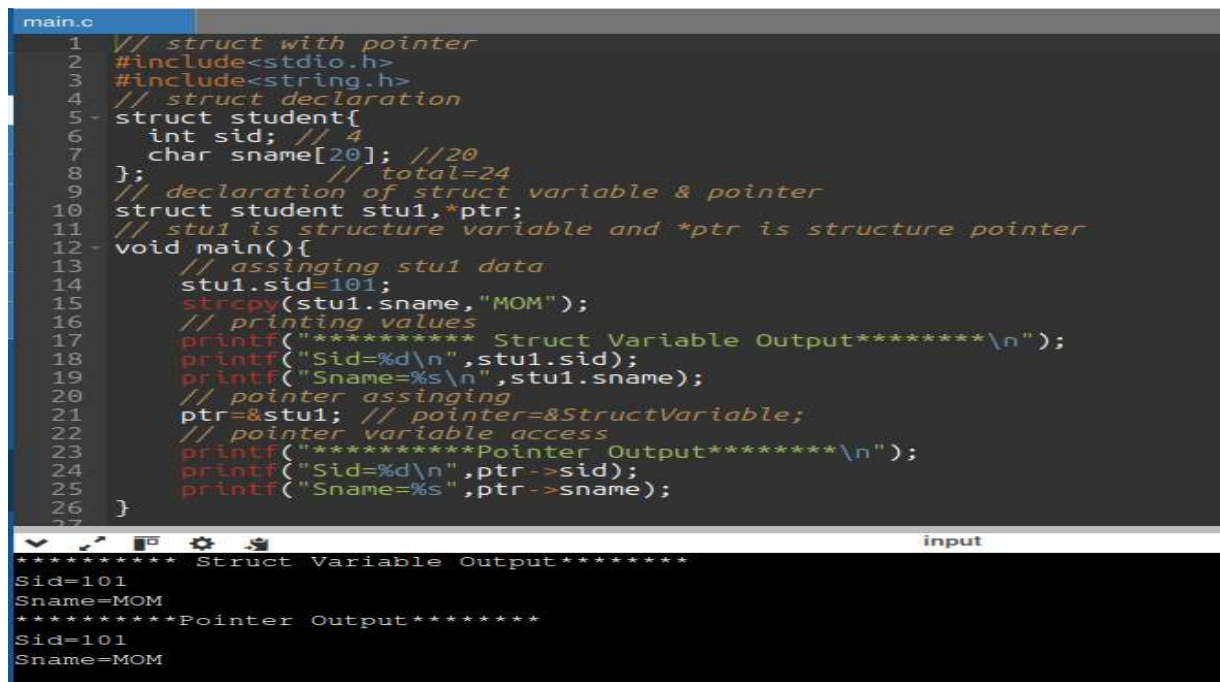
Address of ptr=543A7AAC

## 11. Member Access Operators

Used with structures and unions:

- . (Dot operator): structVar.member
- -> (Arrow operator): ptr->member

### Example:



```
main.c
1 // struct with pointer
2 #include<stdio.h>
3 #include<string.h>
4 // struct declaration
5 struct student{
6     int sid; // 4
7     char sname[20]; //20
8 }; // total=24
9 // declaration of struct variable & pointer
10 struct student stu1,*ptr;
11 // stu1 is structure variable and *ptr is structure pointer
12 void main(){
13     // assinging stu1 data
14     stu1.sid=101;
15     strcpy(stu1.sname,"MOM");
16     // printing values
17     printf("***** Struct Variable Output*****\n");
18     printf("Sid=%d\n",stu1.sid);
19     printf("Sname=%s\n",stu1.sname);
20     // pointer assinging
21     ptr=&stu1; // pointer=&StructVariable;
22     // pointer variable access
23     printf("*****Pointer Output*****\n");
24     printf("Sid=%d\n",ptr->sid);
25     printf("Sname=%s",ptr->sname);
26 }
27
```

input

```
***** Struct Variable Output*****
Sid=101
Sname=MOM
*****Pointer Output*****
Sid=101
Sname=MOM
```

**Keyword** is a predefined or reserved word in C library with a fixed meaning and used to perform an internal operation. C Language supports 32 keywords.

32 Keywords in C Language

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

### **Type conversion and casting:**

Type conversion and casting are essential concepts in C programming that allow you to convert variables from one type to another. Here's a detailed explanation:

#### **Type Conversion**

**Type Conversion** refers to the process of converting one data type to another. In C, there are two main types of type conversion:

##### ***A) Implicit Type Conversion (Automatic Type Conversion)***

Implicit type conversion is performed automatically by the compiler when an operation involves different data types. This is also known as type promotion.

#### **Rules of Implicit Type Conversion:**

- **Integer Promotion:** Smaller integer types (e.g., char, short) are promoted to int if they are used in expressions.
- **Arithmetic Conversions:** When performing arithmetic operations, if the operands have different types, they are converted to a common type according to the usual arithmetic conversions.

#### **Example:**

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

```
int main() {
    int a = 5;
    float b = 4.2;
```

```
float result;

result = a + b; // 'a' is implicitly converted to float
printf("Result: %f\n", result); // Output: Result: 9.200000

return 0;
}
```

In this example, a is automatically converted to float to match the type of b during the addition operation.

### ***B) Explicit Type Conversion (Type Casting)***

Explicit type conversion, or type casting, is performed manually by the programmer using a cast operator. It allows you to convert a variable to a specific type.

#### **Syntax:**

(type) expression

#### **Example:**

```
#include <stdio.h>

void main() {
    float pi = 3.14;
    int integer_part;
    integer_part = (int)pi; // Explicitly casting float to int
    printf("Integer part: %d\n", integer_part); // Output: Integer part: 3
}
```

In this example, the float pi is cast to an integer, which results in the loss of the fractional part.

## **2. Type Casting**

**Type Casting** is the process of converting a variable from one data type to another using the cast operator. It is useful when you need to convert between incompatible types or control the precision and range of numeric data.

### **Types of Type Casting:**

#### ***2.1 Implicit Casting (Automatic Conversion)***

Occurs automatically when different data types are used together, as shown in the implicit type conversion example above.

## ***2.2 Explicit Casting***

Requires the use of the cast operator to convert a variable to a different type.

### **Examples:**

- **Casting Between Numeric Types:**

```
#include <stdio.h>

int main() {
    double x = 9.876;
    int y;

    y = (int)x; // Cast double to int
    printf("Value of y: %d\n", y); // Output: Value of y: 9

    return 0;
}
```

- **Casting Pointers:**

```
#include <stdio.h>

int main() {
    int num = 10;
    void *ptr;

    ptr = &num; // Assign address of num to void pointer
    printf("Address stored in ptr: %p\n", ptr);

    int *int_ptr = (int *)ptr; // Cast void pointer back to int pointer
    printf("Value pointed to by int_ptr: %d\n", *int_ptr); // Output: Value pointed to by int_ptr: 10

    return 0;
}
```

- **Casting Between int and char:**

```
#include <stdio.h>

int main() {
    char ch = 'A';
    int ascii_val;

    ascii_val = (int)ch; // Cast char to int
    printf("ASCII value of %c: %d\n", ch, ascii_val); // Output: ASCII value of A: 65

    return 0;
}
```

}

### Algorithmic Approach

**Algorithmic Approach:** An algorithmic approach to problem-solving involves designing a step-by-step procedure or a set of rules to achieve a specific goal or solve a problem. It typically includes:

1. **Defining the Problem:** Clearly specify the problem or task you want to solve.
2. **Designing an Algorithm:** Create a sequence of steps or instructions to solve the problem. This can be done using pseudo code, flowcharts, or detailed descriptions.
3. **Implementing the Algorithm:** Convert the algorithm into a programming language to create a working program.
4. **Testing and Debugging:** Run the program with various inputs to ensure it works correctly and fix any issues that arise.
5. **Optimizing:** Improve the algorithm to make it more efficient in terms of time and space.

### Characteristics of an Algorithm

An algorithm should have the following characteristics:

1. **Finiteness:** The algorithm must terminate after a finite number of steps.
2. **Definiteness:** Each step of the algorithm must be clear and unambiguous.
3. **Input:** The algorithm should have zero or more inputs, which are the data it operates on.
4. **Output:** The algorithm should produce at least one output, which is the result or solution to the problem.
5. **Effectiveness:** Each step should be basic enough to be performed exactly and in a finite amount of time.

### Problem Solving Strategies

#### 1. Top-Down Approach:

- **Description:** Break down a problem into smaller, more manageable sub-problems or modules. Start by defining the high-level tasks and then decompose them into smaller tasks.
- **Example:** Top-down approach might involve starting with an overall system design and then breaking it down into smaller modules or functions.

The top-down approach involves breaking down a complex problem into simpler, more manageable sub-problems. You start with the broad, high-level view of the problem and progressively decompose it into smaller, more specific parts.

*Steps:*

1. **Define the Problem:** Start with the overall problem or goal.
2. **Decompose the Problem:** Break the main problem into smaller sub-problems or components.
3. **Solve Sub-Problems:** Address each sub-problem individually.
4. **Integrate Solutions:** Combine the solutions of the sub-problems to solve the original problem.

*Advantages:*

- **Clear Focus:** Helps in understanding the overall structure of the problem and its components.
- **Organized:** Facilitates a systematic approach to problem-solving.
- **Modular:** Solutions to sub-problems can often be reused in other contexts.

*Disadvantages:*

- **Complexity:** Can become cumbersome if the decomposition is too detailed or if the problem is too complex.
- **Overhead:** May involve additional overhead in managing the multiple layers of sub-problems.

2. **Bottom-Up Approach:**

- **Description:** Start with the details or smaller sub-problems and build up to the larger problem. Develop and test each module or component individually before integrating them into a complete solution.
- **Example:** In programming, a bottom-up approach might involve developing individual functions or classes first and then integrating them into a larger system.

The bottom-up approach focuses on solving small, basic problems first and then building up to the solution of the overall problem. It starts with the details and works towards the higher-level abstraction.

*Steps:*

1. **Identify Basic Components:** Begin by solving the simplest, most fundamental problems or creating basic building blocks.
2. **Build Upwards:** Integrate these basic components to form more complex structures or solve higher-level problems.
3. **Assemble the Whole:** Combine the pieces to address the larger problem.

*Advantages:*

- **Practical:** Useful when specific solutions are known or when dealing with well-defined

components.

- **Flexibility:** Allows for incremental development and refinement.
- **Iterative:** Can adapt and change as you learn more about the components and their interactions.

**Disadvantages:**

- **Integration Challenges:** May face difficulties in integrating the components if not properly designed from the start.
- **Less Structure:** Can be less organized, making it harder to see the overall problem structure early on.

Basis for comparison	Top-down Approach	Bottom-up Approach
Basic	Breaks the massive problem into smaller subproblems.	Solves the fundamental low-level problem and integrates them into a larger one.
Process	Submodules are solitarily analysed.	Examine what data is to be encapsulated, and implies the concept of information hiding.
Communication	Not required in the top-down approach.	Needs a specific amount of communication.
Redundancy	Contain redundant information.	Redundancy can be eliminated.
Programming languages	Structure/procedural oriented programming languages (i.e. C) follows the top-down approach.	Object-oriented programming languages (like C++, Java, etc.) follows the bottom-up approach.

## Time and Space Complexities of Algorithms

### Time Complexity:

- **Definition:** Measures the amount of time an algorithm takes to complete as a function of the size of the input. It gives an idea of the algorithm's efficiency.
- **Common Classes:**
  - **$O(1)$ :** Constant time complexity. The execution time is independent of the input size.
  - **$O(\log n)$ :** Logarithmic time complexity. The execution time increases logarithmically with the input size.
  - **$O(n)$ :** Linear time complexity. The execution time increases linearly with the

input size.

- **$O(n^2)$** : Quadratic time complexity. The execution time increases quadratically with the input size.
- **$O(2^n)$** : Exponential time complexity. The execution time grows exponentially with the input size.

### **Space Complexity:**

- **Definition:** Measures the amount of memory an algorithm uses as a function of the size of the input. It provides insight into the memory efficiency of the algorithm.
- **Common Classes:**
  - **$O(1)$** : Constant space complexity. The memory usage is fixed and does not depend on the input size.
  - **$O(n)$** : Linear space complexity. The memory usage increases linearly with the input size.
  - **$O(n^2)$** : Quadratic space complexity. The memory usage increases quadratically with the input size.

### **Example 1: Linear Search**

**Problem:** Find if a target value exists in an unsorted array.

#### **C Code:**

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

bool linear_search(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return true;
        }
    }
    return false;
}

int main() {
    int arr[] = {3, 5, 7, 9, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;

    if (linear_search(arr, size, target)) {
        printf("Target found!\n");
    } else {
        printf("Target not found.\n");
    }
}
```



```
return 0;
}
```

### **Complexity Analysis:**

- **Time Complexity:**
  - **Best Case:**  $O(1)$  – If the target is at the first position.
  - **Worst Case:**  $O(n)$  – If the target is at the last position or not present.
  - **Average Case:**  $O(n)$  – On average, the search may check half of the elements.
- **Space Complexity:**  $O(1)$  – Constant space usage, only a few variables are used.

### **Example 2: Binary Search**

**Problem:** Find if a target value exists in a sorted array.

#### **C Code:**

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

bool binary_search(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return true;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return false;
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;

    if (binary_search(arr, size, target)) {
        printf("Target found!\n");
    } else {
```

```
    printf("Target not found.\n");
}

return 0;
}
```

### **Complexity Analysis:**

- **Time Complexity:**
  - **Best Case:**  $O(1)$  – If the target is at the middle position.
  - **Worst Case:**  $O(\log n)$  – The search space is halved each time.
  - **Average Case:**  $O(\log n)$  – On average, the search will require logarithmic steps.
- **Space Complexity:**  $O(1)$  – Constant space usage for variables.

### **Example 3: Bubble Sort**

**Problem:** Sort an array of integers using Bubble Sort.

#### **C Code:**

```
#include <stdio.h>

void bubble_sort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
bubble_sort(arr, size);

printf("Sorted array: \n");
print_array(arr, size);

return 0;
}
```

### Complexity Analysis:

- **Time Complexity:**
  - **Best Case:**  $O(n)$  – If the array is already sorted (optimizations can be added to detect this case).
  - **Worst Case:**  $O(n^2)$  – In the case of a reverse-sorted array, where each pair of elements needs to be compared and swapped.
  - **Average Case:**  $O(n^2)$  – Generally requires quadratic time for most cases.
- **Space Complexity:**  $O(1)$  – Bubble Sort is an in-place algorithm requiring only a constant amount of additional space.

### Note:-

- **Linear Search:**  $O(n)$  time complexity and  $O(1)$  space complexity.
- **Binary Search:**  $O(\log n)$  time complexity and  $O(1)$  space complexity.
- **Bubble Sort:**  $O(n^2)$  time complexity and  $O(1)$  space complexity.

### Summary

- **Algorithmic Approach:** Design and implement step-by-step procedures for problem-solving.
- **Characteristics of Algorithms:** Finiteness, definiteness, input, output, and effectiveness.
- **Problem-Solving Strategies:** Top-down (high-level to low-level) and Bottom-up (low-level to high-level).
- **Complexity Analysis:** Assess time (speed) and space (memory) efficiency of algorithms using Big O notation.