Module 1
Introduction to Computers

1.1 Computer

- ✓ A computer can be defined as an electronic device that is designed to accept data, perform the required mathematical and logical operations at high speed, and output the result.

1.2 Characteristics of Computers

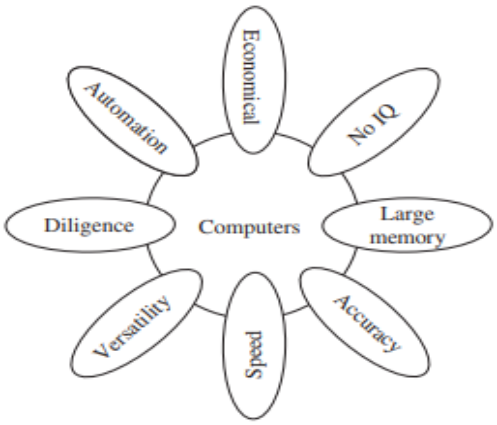


Figure 1.1. Characteristics of computers

1. Speed

- ✓ Computers can perform millions of operations per second.
- ✓ The speed of computers is usually given in nanoseconds and picoseconds, where 1 nanosecond = 1×10^{-9} seconds and 1 picosecond = 1×10^{-12} seconds.

2. Accuracy

- ✓ A computer is a very fast, reliable, and robust electronic device.
- ✓ It always gives accurate results, provided the correct data and set of instructions are input to it.
- ✓ This clearly means that the output generated by a computer depends on the given instructions and input data.
- ✓ If the input data is wrong, then the output will also be erroneous. In computer terminology, this is known as garbage-in, garbage-out (GIGO).

3. Automation

- ✓ Besides being very fast and accurate, computers are automatable devices that can perform a task without any user intervention.
- ✓ The user just needs to assign the task to the computer, after which it automatically controls different devices attached to it and executes the program instructions.

4. Diligence

- ✓ Computers never get tired of a repetitive task. It can continually work for hours without creating errors.
- ✓ Even if a large number of executions need to be executed, each and every execution requires the same duration, and is executed with the same accuracy.

5. Versatile

- ✓ Versatility is the quality of being flexible. Today, computers are used in our daily life in different fields.
- ✓ For example, they are used as personal computers (PCs) for home use, for business-oriented tasks, weather forecasting, space exploration, teaching, railways, banking, medicine, and so on, indicating that computers can perform different tasks simultaneously.

6. Memory

- ✓ Computers have internal or primary memory (storage space) as well as external or secondary memory.
- ✓ The computer stores a large amount of data and programs in the secondary storage space.
- ✓ The stored data and programs can be retrieved and used whenever required.
- ✓ Secondary memory is the key for data storage. Some examples of secondary devices include floppy disks, optical disks (CDs and DVDs), hard disk drives (HDDs), and pen drives.

7. No IQ

- ✓ Although the trend today is to make computers intelligent by inducing artificial intelligence (AI) in them, they still do not have any decision-making abilities of their own. They need guidance to perform various tasks.

8. Economical

- ✓ Today, computers are considered as short-term investments for achieving long-term gains. Using computers also reduces manpower requirements and leads to an elegant and efficient way of performing various tasks.
- ✓ Hence, computers save time, energy, and money. When compared to other systems, computers can do more work in lesser time.
- ✓ For example, using the conventional postal system to send an important document takes at least two to three days, whereas the same information when sent using the Internet (e-mail) will be delivered instantaneously.

1.3 Stored Program Concept

- ✓ All digital computers are based on the principle of stored program concept.
- ✓ The following are the key characteristic features of this concept:
  - Before any data is processed, **instructions are read into memory**.
  - **Instructions are stored in the computer's memory** for execution.
  - **Instructions are stored in binary form** (using binary numbers—only 0s and 1s).
  - **Processing starts with the first instruction** in the program, which is copied into a control unit circuit. The control unit executes the instructions.
  - **Instructions written by the users are performed sequentially until there is a break in the current flow.**
  - **Input/Output** and processing operations are performed simultaneously. While data is being read/written, the central processing unit (CPU) executes another program in the memory that is ready for execution.

1.3.1 Types of Stored Program Computers

1. Shared Memory Architecture

- ✓ A computer with a Von Neumann architecture stores data and instructions in the same memory.
- ✓ There is a serial machine in which data and instructions are selected one at a time.
- ✓ Data and instructions are transferred to and from memory through a shared data bus. Since there is a single bus to carry data and instructions, process execution becomes slower.

2. Distributed Memory Architecture

- ✓ Later Harvard University proposed a stored program concept in which there was a separate memory to store data and instructions.
- ✓ Instructions are selected serially from the instruction memory and executed in the processor. When an instruction needs data, it is selected from the data memory. Since there are separate memories, execution becomes faster.

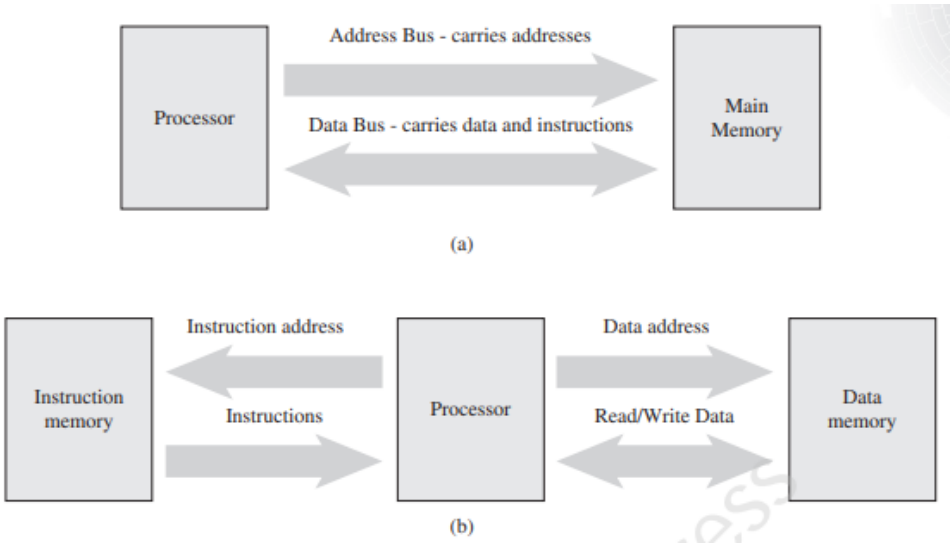


Figure 1.2. Von Neumann architecture (a) Shared memory for instructions and data (b) Separate memories for instructions and data

1.4 History of Computers

✓ Timeline of Developments

300 BC: The abacus was an early aid for mathematical computations and was designed to aid human's memory while performing calculations.

1822: Charles Babbage designed a steam-driven calculating machine that could compute tables of numbers.

1890: a punched card system was designed for the calculations.

1936: Turing machine were designed, which were capable of computing anything that is computable.

1941: It was the first time a computer could store information in its main memory.

1943–1944:

- Electronic Numerical Integrator and Calculator (ENIAC) were built.
- It is considered as the grandfather of digital computers.
- It filled a 20 × 40 feet room and had 18,000 vacuum tubes.

1946: UNIVAC was designed, which was the first commercial computer for business and government applications.

1947: vacuum tubes in computers were replaced by transistors.

1953: Developed the first computer language COBOL.

1954: The FORTRAN programming language was developed.

1958: invented integrated circuit, which is commonly known as the computer chip.

1964: developed a prototype of the modern computer, with a mouse and a graphical user interface (GUI). This was a remarkable achievement as it shifted computers from a specialized machine for scientists and mathematicians to general public.

1969: Unix operating system was developed at Bell Labs. It was written in the C programming language and was designed to be portable across multiple platforms.

1970: DRAM chip was introduced by Intel.

1971: invented the floppy disk which allowed data to be shared among computers.

1973: developed Ethernet for connecting multiple computers and other hardware.

1974–1977: Personal computers started becoming popular.

1975: Paul Allen and Bill Gates started writing software using the new BASIC language. They both formed their own software company, Microsoft.

1976: Apple Computers and developed Apple I, the first computer with a single-circuit board.

1977: Apple II was launched that offered colour graphics and incorporated an audio cassette drive for storage.
1978: a word processor application, was released by MicroPro International.
1979: the first computerized spreadsheet program for personal computers, was unveiled.
1981: The first IBM personal computer was introduced that used Microsoft's MS-DOS operating system. The term PC was popularized.
1983: The first laptop was introduced.
1985: Microsoft announced Windows as a new operating system.
1986: Compaq introduced Deskpro 386 in the market, which was a 32-bit architecture machine that provides speed comparable to mainframes.
1990: Invented the World Wide Web with HTML as its publishing language.
1993: The Pentium microprocessor introduced the use of graphics and music on PCs.
1994: PC games became popular.
1996: developed the Google search engine at Stanford University.
1999: The term Wi-Fi was introduced when users started connecting to the Internet without wires.
2001: Apple introduced Mac OS X operating system, which had protected memory architecture and pre-emptive multi-tasking, among other benefits. To stay competitive, Microsoft launched Windows XP.
2003: The first 64-bit processor, AMD's Athlon 64, was brought into the consumer market.
2004: Mozilla released Firefox 1.0 and in the same year Facebook, a social networking site, was launched.
2005: YouTube, a video sharing service, was launched. In the same year, Google acquired Android, a Linux-based mobile phone operating system.
2006: Apple introduced MacBook Pro, its first Intelbased, dual-core mobile computer.
2007: Apple released iPhone, which brought many computer functions in the smartphone.
2009: Microsoft launched Windows 7 in which users could pin applications to the taskbar.
2010: Apple launched iPad, which revived the tablet computer segment.
2011: Google introduced Chromebook, a laptop that runs on the Google Chrome operating system.
2015: Apple released the Apple Watch. In the same year, Microsoft launched Windows 10.

1.4.1 Generations of Computers

1. First Generation (1942–1955)

Hardware Technology

First generation computers were manufactured using thousands of ***vacuum tubes***.

Memory

- **Electromagnetic** relay was used as **primary memory** and
- **Punched cards** were used to **store data and instructions**.

Software Technology

- Programming was done in machine or assembly language.

Used for Scientific applications

Examples ENIAC, EDVAC, EDSAC, UNIVAC I, IBM 701

Highlights

- They were the fastest calculating device.
- Computers were too bulky and required a complete room for storage
- Highly unreliable as vacuum tubes emitted a large amount of heat and burnt frequently

- Required air-conditioned rooms for installation
- Costly
- Difficult to use
- Required constant maintenance because vacuum tubes used filaments that had limited life time. Therefore, these computers were prone to frequent hardware failures.

2. Second Generation (1955–1964)

Hardware Technology

- Second generation computers were manufactured using ***transistors.***
- Transistors were reliable, powerful, cheaper, smaller, and cooler than vacuum tubes.

Memory

- Magnetic core memory was used as primary memory;
- Magnetic tapes and magnetic disks were used to store data and instructions.
- These computers had faster and larger memory than the first generation computers.

Software Technology

- Programming was done in high level programming languages. Batch operating system was used.

Used for Scientific and commercial applications

Examples Honeywell 400, IBM 7030, CDC 1604, UNIVAC LARC

Highlights

- Faster, smaller, cheaper, reliable, and easier to use
- They consumed 1/10th the power consumed by first generation computers
- Bulky in size
- Dissipated less heat than first generation computers
- Costly
- Difficult to use.

3. Third Generation (1964–1975)

Hardware Technology

- Third generation computers were manufactured using integrated chips (ICs).
- ICs consist of several components such as transistors, capacitors, and resistors on a single chip to avoid wired interconnections between components.
- These computers used SSI and MSI technology. Minicomputers came into existence.

Memory

- Larger magnetic core memory was used as primary memory; larger capacity magnetic tapes and magnetic disks were used to store data and instructions.

Used for Scientific, commercial, and interactive online applications.

Highlights

- Faster, smaller, cheaper, reliable, and easier to use than the second generation computers
- They consumed less power than second generation computers.
- Bulky in size and required a complete room for installation
- Dissipated less heat than second generation computers
- Costly

- Easier to use and upgrade

Software Technology

- Programming was done in high level programming languages such as FORTRAN, COBOL, Pascal, and BASIC.
- Time sharing operating system was used.
- Software was separated from the hardware.

4. Fourth Generation (1975–1989)

Hardware Technology

- Fourth generation computers were manufactured using ICs with LSI (Large Scale Integrated) and later with VLSI technology (Very Large Scale Integration).
- Microcomputers came into existence.
- Use of personal computers became widespread.
- High speed computer networks in the form of LANs, WANs, and MANs started growing.

Memory

- Semiconductor memory was used as primary memory; large capacity magnetic disks were used as built in secondary memory.
- Magnetic tapes and floppy disks were used as portable storage devices.

Software Technology

- Programming was done in **high level programming language such as C and C++.**
- **Graphical User Interface** (GUI) based operating system (e.g. Windows) was introduced. It had **icons and menus** among other features to allow computers to be used as a general purpose machine by all users.
- UNIX was also introduced as an open source operating system. Apple Mac OS and MS DOS were also released during this period.
- All these operating systems had multi-processing and multiprogramming capabilities.

Used for Scientific, commercial, interactive online, and network applications.

Examples IBM PC, Apple II, TRS-80, VAX 9000, CRAY1, CRAY-2, CRAY-X

Highlights

- Faster, smaller, cheaper, powerful, reliable, and easier to use than the previous generation computers.

5. Fifth Generation (1989–Present)

Hardware Technology

- Fifth generation computers are manufactured using ICs with ULSI (Ultra Large Scale Integrated) technology.
- The use of Internet became widespread and very powerful mainframes, desktops, portable laptops, and smartphones are being used commonly.
- Supercomputers use parallel processing techniques.

Memory

- Semiconductor memory is used as primary memory;
- large capacity magnetic disks are used as built-in secondary memory.
- portable storage devices like optical disks and USB flash drives are used.

Software Technology

- Programming is done in high-level programming languages such as Java, Python, and C#.
- Graphical User Interface (GUI)-based operating systems such as Windows, Unix, Linux, Ubuntu, and Apple Mac are being used.
- These operating systems are more powerful and user friendly than the ones available in the previous generations.

Used for Scientific, commercial, interactive online, multimedia (graphics, audio, video), and network applications.

Examples IBM notebooks, Pentium PCs, SUM workstations, IBM SP/2, Param supercomputer

Highlights

- Faster, smaller, cheaper, powerful, reliable, and easier to use than the previous generation computers.
- Speed of microprocessors and the size of memory are growing rapidly.
- High-end features available on the microprocessors
- They consume less power than computers of prior generations.

1.5 Classification of Computers

✓ Computers can be broadly classified into four categories based on their speed, amount of data that they can process, and price:

- Supercomputers
- Mainframe computers
- Minicomputers
- Microcomputers

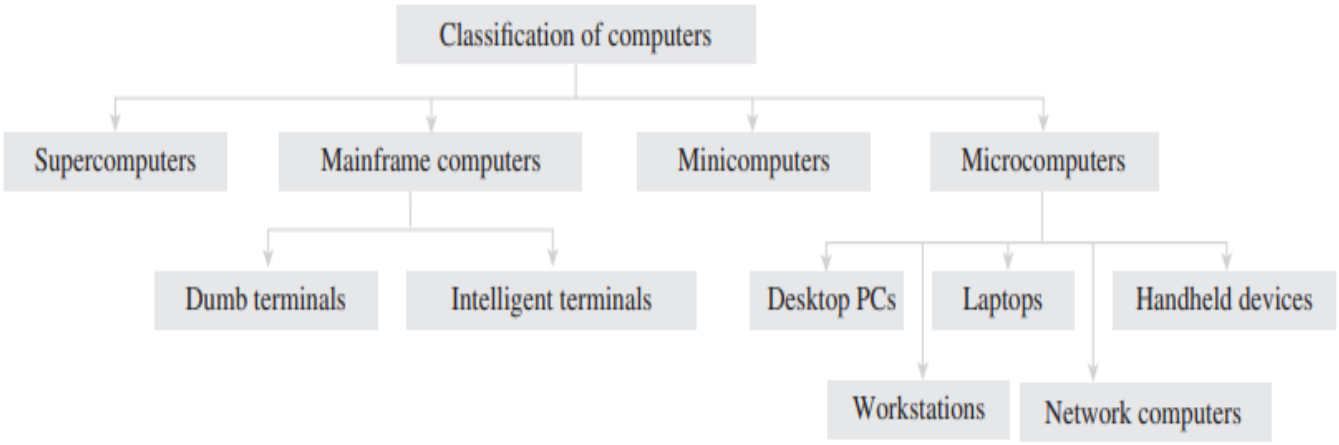


Figure 1.3. Classification of computers

1. Supercomputers

✓ The supercomputer is the fastest, most powerful, and most expensive computer.
✓ Process large amounts of data and to solve complex scientific problems.
✓ Supercomputers use parallel processing technology and can perform more than one trillion calculations in a second.
✓ A single supercomputer can support thousands of users at the same time.
✓ Such computers are mainly used for weather forecasting, nuclear energy research, aircraft design, automotive design, online banking, controlling industrial units, etc.

2. Mainframe Computers

- ✓ Mainframe computers are large-scale computers (but smaller than supercomputers).
- ✓ These are very expensive and need a very large clean room with air conditioning, thereby making them very costly to deploy.
- ✓ Support multiple processors.
- ✓ The two types of terminals that can be used with mainframe systems are as follows:

1. Dumb Terminals

- ✓ Dumb terminals consist of only a monitor and a keyboard (or mouse). They do not have their own CPU and memory and use the mainframe system's CPU and storage devices.

2. Intelligent Terminals:

- ✓ Intelligent terminals have their own processor and thus can perform some processing operations.
- ✓ Usually, PCs are used as intelligent terminals to facilitate data access and other services from the mainframe system.
- ✓ Mainframe computers are typically used as servers on the World Wide Web.
- ✓ They are also used in organizations such as banks, airline companies, and universities, where a large number of users frequently access the data stored in their databases.

3. Minicomputers

- ✓ Minicomputers are smaller, cheaper, and slower than mainframes.
- ✓ They are called minicomputers because they were the smallest computer of their times. Also known as midrange computers.
- ✓ Minicomputers are widely used in business, education, hospitals, government organizations, etc. While some minicomputers can be used only by a single user, others are specifically designed to handle multiple users simultaneously.
- ✓ Minicomputers can also be used as servers in a networked environment.

4. Microcomputers

- ✓ Microcomputers, commonly known as PCs, are very small and cheap.
- ✓ Many computer hardware companies copied this design and termed their microcomputers as PC-compatible.
- ✓ PCs can be classified into the following categories:
  Desktop PCs, Laptops, Workstations, Network Computers and Handheld Computers.
  (i) Desktop PCs A desktop PC is the most popular model of PCs.
  (ii) Laptops: Laptops operate on a battery. The memory and storage capacity of a laptop is almost equivalent to that of a desktop computer. laptops have the same features and processing speed as the most powerful PCs.
  (iii) Workstations: Workstation computers have advanced processors, more RAM and storage capacity than PCs.
  (iv) Network Computers: Network computers have less processing power, memory, and storage than a desktop computer. Some network computers do not have any storage space and merely rely on the network's server for data storage and processing tasks
  (v) Handheld Computers: Handheld computers are very small in size, and hence they have small-sized screens and keyboards. Some examples of handheld computers are as follows: Smartphones, Tablet PCs.
  a. Smartphones: These days, cellular phones are web-enabled telephones. Such phones are also known as smartphones because, in addition to basic phone capabilities, they also facilitate the users to access the Internet and send e-mails, edit Word documents, generate an Excel sheet, create a presentation, and lots more.
  b. Tablet PCs: A tablet PC is a computing device that is smaller than a laptop, but bigger than a smartphone. Features such as user-friendly interface, portability, and touch screen have made them very popular in the last few years.

Uses The following are the uses of Tablet PCs:
• View presentations
• Videoconferencing
• Reading e-books, e-newspaper
• Watching movies
• Playing games
• Sharing pictures, video, songs, documents, etc.
• Browsing the Internet
• Keeping in touch with friends and family on popular social networks, sending emails
• Business people use them to perform tasks such as editing a document, exchanging documents, taking notes, and giving presentations
• Tablets are best used in crowded places such as airports and coffee shops.

1.6 Applications of Computers

1. Word processing

- ✓ Word processing software enables users to read and write documents.
- ✓ Users can also add images, tables, and graphs for illustrating a concept.
- ✓ The software automatically corrects spelling mistakes and includes copy–paste features.

2. Internet

- ✓ The Internet is a network of networks that connects computers all over the world.
- ✓ It gives the user access to an enormous amount of information, much more than available in any library.
- ✓ Using e-mail, the user can communicate in seconds with a person who is located thousands of miles away.
- ✓ Chat software enables users to chat with another person in real-time.
- ✓ Video conferencing tools are becoming popular.

3. Digital video or audio composition

- ✓ Computers make audio or video composition and editing very simple.
- ✓ Graphics engineers use computers for developing short or full-length films and creating 3-D models and special effects in science fiction and action movies.

4. Desktop publishing

- ✓ Desktop publishing software enables us to create page layouts for entire books.

5. e-Business

- ✓ e-Business or electronic business is the process of conducting business via the Internet.
- ✓ This may include buying and selling of goods and services using computers and the Internet.
- ✓ Companies today use e-commerce applications for marketing, transaction, processing, and product and customer services processing.

  Business-to-consumer or B2C: In this form of electronic commerce, business companies deploy their websites on the Internet to sell their products and services to the customers. On their websites, they provide features such as catalogues, interactive order processing system, secure electronic payment system, and online customer support.

  Business-to-business or B2B: This type of electronic commerce involves business transactions performed between business partners (customers are not involved). For example, companies use computers and networks (in the form of extranets) to order raw materials from their suppliers. Companies can also use extranets to supply their products to their dealers.

  Consumer-to-consumer or C2C: This type of electronic commerce enables customers to carry business transactions among themselves. For example, on auction websites, a customer sells his/her product which is purchased by another customer.

Electronic banking Electronic banking, also known as cyber banking or online banking, supports various banking activities conducted from home, a business, or on the road instead of a physical bank location.

6. Bioinformatics

- ✓ Bioinformatics is the application of computer technology to manage large amount of biological information.
- ✓ Computers are used to collect, store, analyse, and integrate biological and genetic information to facilitate gene-based drug discovery and development.
- ✓ Bioinformatics is an interdisciplinary field of molecular biology, computer science, statistics, and mathematics.

7. Health care

- ✓ Computers have also become a necessary device in the health care industry. The following are areas in which computers are extensively used in the health care industry:
  Storing records: computers are first and foremost used to store the medical records of patients.
  Surgical procedures Computers are used for certain surgical procedures. They enable the surgeon to use computer to control and move surgical instruments in the patient's body for a variety of surgical procedures.
  Better diagnosis and treatment Computers help physicians make better diagnoses and recommend treatments.

8. Geographic Information System and Remote Sensing

- ✓ A geographic information system (GIS) is a computer based tool for mapping and analysing earth.
- ✓ It integrates database operations and statistical analysis to be used with maps.
- ✓ Remote sensing is a sub-field of geography, which can be applied in the following areas to collect data of dangerous or inaccessible areas for the following: Monitoring deforestation, Studying features of glaciers, etc.

9. Meteorology

- ✓ Meteorology is the study of the atmosphere. Meteorology has applications in many diverse fields. Some of the applications include the following:
- ✓ Weather forecasting It includes application of science and technology to predict the state of the atmosphere (temperature, precipitation, etc.) for a future time and a given location.
- ✓ Aviation meteorology Aviation meteorology studies the impact of weather on air traffi c management.
- ✓ Agricultural meteorology Agricultural meteorology deals with the study of effects of weather and climate on plant distribution, crop yield, water-use efficiency, plant and animal development.
- ✓ Nuclear meteorology Nuclear meteorology studies the distribution of radioactive aerosols and gases in the atmosphere. Maritime meteorology Maritime meteorology is the study of air and wave forecasts for ships operating at sea.
- ✓ Multimedia and Animation Multimedia and animation that combines still images, moving images, text, and sound in meaningful ways is one of most powerful aspects of computer technology. We all have seen cartoon movies, which are nothing but an example of computer animation.

10. Legal System

- ✓ Lawyers use computers to look through millions of individual cases and find whether similar or parallel cases have been approved, denied, criticized, or overruled in the past.

11. Retail Business

✓ Computers are used in retail shops to enter orders, calculate costs, and print receipts.

12. Sports

✓ In sports, computers are used to compile statistics, identify weak players and strong players by analysing statistics, sell tickets, create training programs and diets for athletes, and suggest game plan strategies based on the competitor's past performance.

13. Travel and Tourism

✓ Computers are used to prepare tickets, monitor the train's or airplane's route, and guide the plane to a safe landing.

14. Simulation

✓ Simulation of automobile crashes or airplane emergency landings is done to identify potential weaknesses in designs without risking human lives.

15. Education

✓ A computer is a powerful teaching aid and can act as another teacher in the classroom. Teachers use computers to develop instructional material.

16. Industry and Engineering

✓ Computers are found in all kinds of industries, such as thermal power plants, oil refineries, and chemical industries, for process control, computer-aided designing (CAD), and computer-aided manufacturing (CAM).

17. Robotics

✓ Robots are computer-controlled machines mainly used in the manufacturing process in extreme conditions where humans cannot work.

18. Decision Support Systems

✓ Computers help managers to analyse their organization's data to understand the present scenario of their business, view the trends in the market, and predict the future of their products.

19. Expert Systems

✓ Expert systems are used to automate the decision-making process in a specific area, such as analysing the credit histories for loan approval and diagnosing a patient's condition for prescribing an appropriate treatment.

1.7 Basic Organization of a Computer

✓ A computer is an electronic device that performs five major operations:

- Accepting data or instructions (input)
- Storing data
- Processing data
- Displaying results (output)
- Controlling and coordinating all operations inside a computer.

Input: This is the process of entering data and instructions (also known as programs) into the computer system. The data and instructions can be entered by using different input devices such as keyboard, mouse, scanner, and trackball.

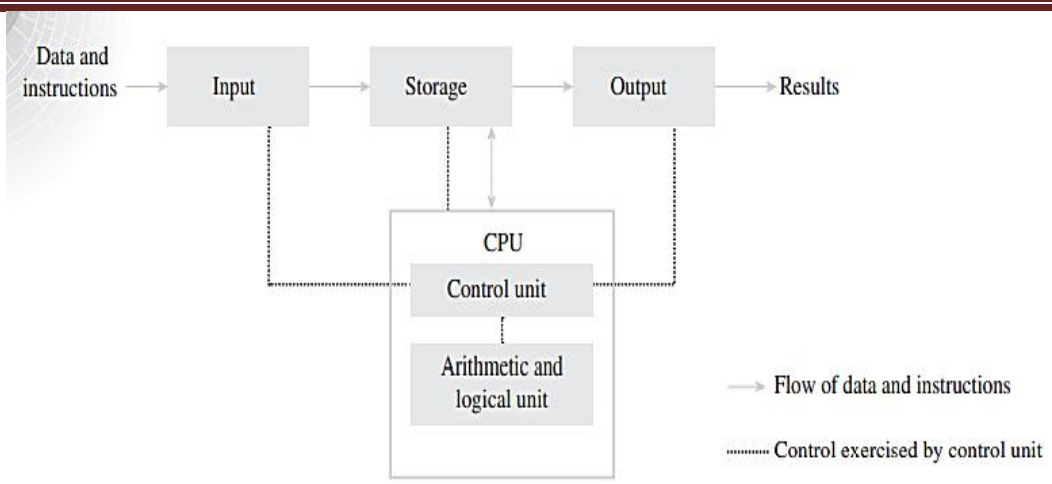


Figure 1.4. Block diagram of a computer

Storage: Storage is the process of saving data and instructions permanently in the computer so that they can be used for processing.
A computer has two types of storage areas:
Primary storage: Primary storage, also known as the main memory, is the storage area that is directly accessible by the CPU at very high speeds. Primary storage space is very expensive and therefore limited in capacity.Another drawback of main memory is that it is volatile in nature; that is, as soon as the computer is switched off, the information stored gets erased. Hence, it cannot be used as a permanent storage of useful data and programs for future use. Example of primary storage is random access memory (RAM).
Secondary storage Also known as auxiliary memory. It is cheaper, non-volatile, and used to permanently store data and programs of those jobs that are not being currently executed by the CPU.

Output: Output is the process of giving the result of data processing to the outside world. The results are given through output devices such as monitor, and printer. Since the computer accepts data only in binary form and the result of processing is also in binary form, the result cannot be directly given to the user. The output devices, therefore, convert the results available in binary codes into a human-readable language before displaying it to the user.

Control: The control unit (CU) manages and controls all the components of the computer system. It is the CU that decides the manner in which instructions will be executed and operations performed. It takes care of the step-by-step processing of all operations that are performed in the computer.
The CPU is a combination of the **arithmetic logic unit (ALU) and the CU**. The CPU is better known as the brain of the computer system because the entire processing of data is done in the ALU, and the CU activates and monitors the operations of other units (such as input, output, and storage) of the computer system.

Processing: The process of performing operations on the data as per the instructions specified by the user (program) is called processing. Data and instructions are taken from the primary memory and transferred to the ALU, which performs all sorts of calculations.

1.8 Motherboard

- ✓ The motherboard, also known as the main board or the parent board is the primary component of a computer. It is used to connect all the components of the computer.

Characteristics of a Motherboard

- ✓ A motherboard can be classified depending on the following characteristics:
  - Form factor
  - Chipset
  - Type of processor socket used
  - Input–Output connectors

Form factor: Form factor refers to the motherboard's geometry, dimensions, arrangement, and electrical requirements.

Integrated components: Some of the motherboard's components are integrated into its printed circuitry. These include the following:

- The chipset is a circuit that controls the majority of the computer's resources such as the bus interface with the processor, cache memory, RAM, and expansion cards.
- CMOS clock and battery
- BIOS
- System bus and expansion bus

Chipset: The chipset is an electronic circuit that basically coordinates data transfers between the different components of the computer.
Some chipsets may include a graphics or audio chip, which makes it unnecessary to install a separate graphics card or sound card.

CMOS clock and battery: The real-time clock (or RTC) is a circuit that is used to synchronize the computer's signals. When the computer is switched off, the power supply stops providing electricity to the motherboard. The CMOS chip is powered by a battery located on the motherboard.

Processor socket: The processor (also called the microprocessor) is the brain of the computer. The processor is characterized by its speed or frequency, which is the rate at which it executes instructions.

RAM connectors RAM is the primary storage area that stores data while the computer is running. However, its contents are erased when the computer is turned off or restarted.

Expansion slots: Expansion slots are compartments into which expansion cards can be inserted. Such cards render new features or enhance the computer's performance.

I/O connectors: The motherboard has a number of input– output sockets on its rear panel, some of which include:

- A serial port to connect some old peripherals
- A parallel port to connect old printers
- USB ports to connect more recent peripherals such as mouse and pen drive.
- RJ45 connector (also known as LAN or Ethernet port) to connect the computer to a network. It corresponds to a network card integrated into the motherboard.
- Video graphics array (VGA) connector to connect a monitor. This connector interfaces with the built-in graphics card.
- Audio plugs that include the line in, line out, and microphone to connect sound speakers, hi-fi system, or microphone. This connector interfaces with the built-in sound card.

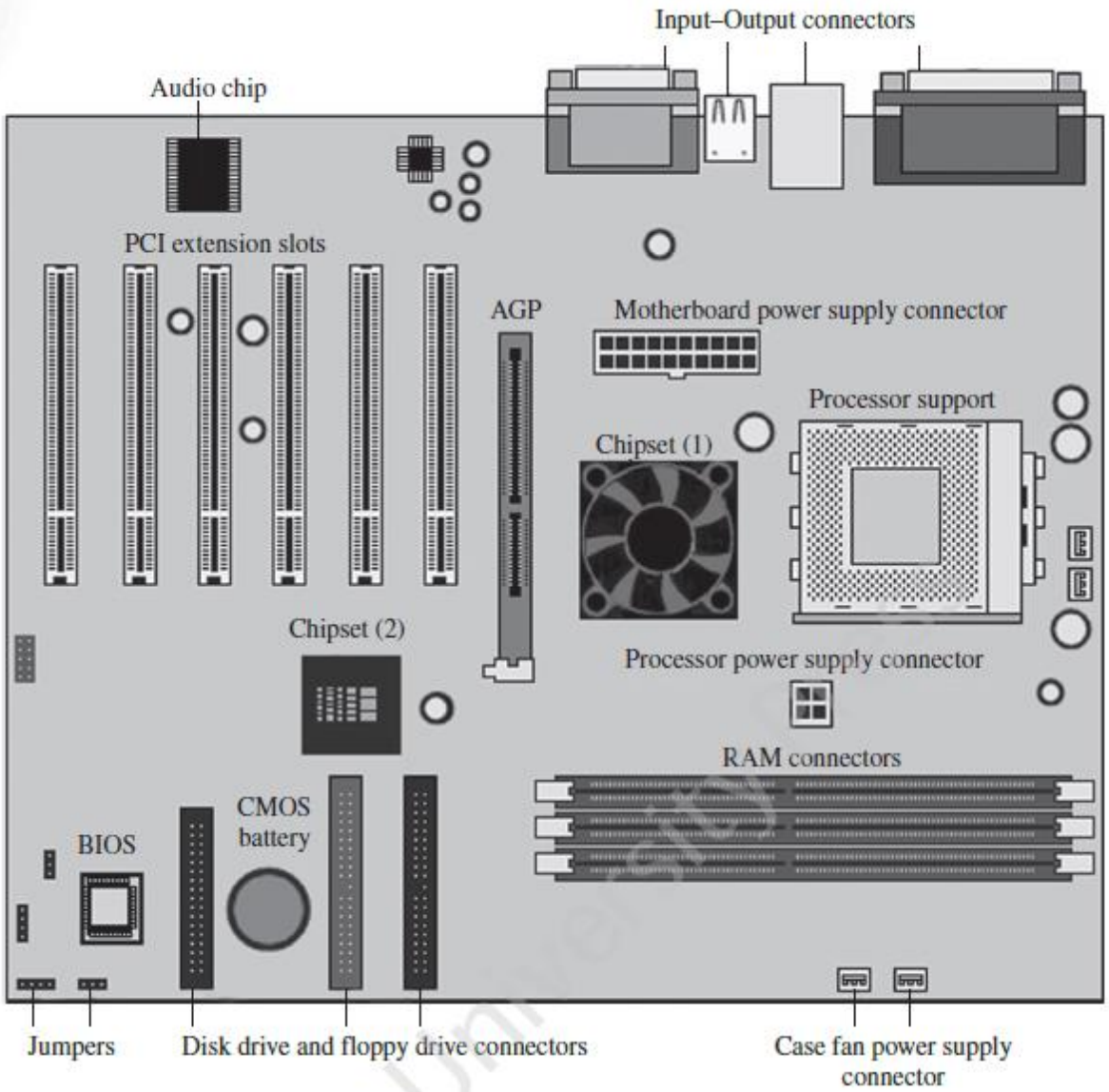


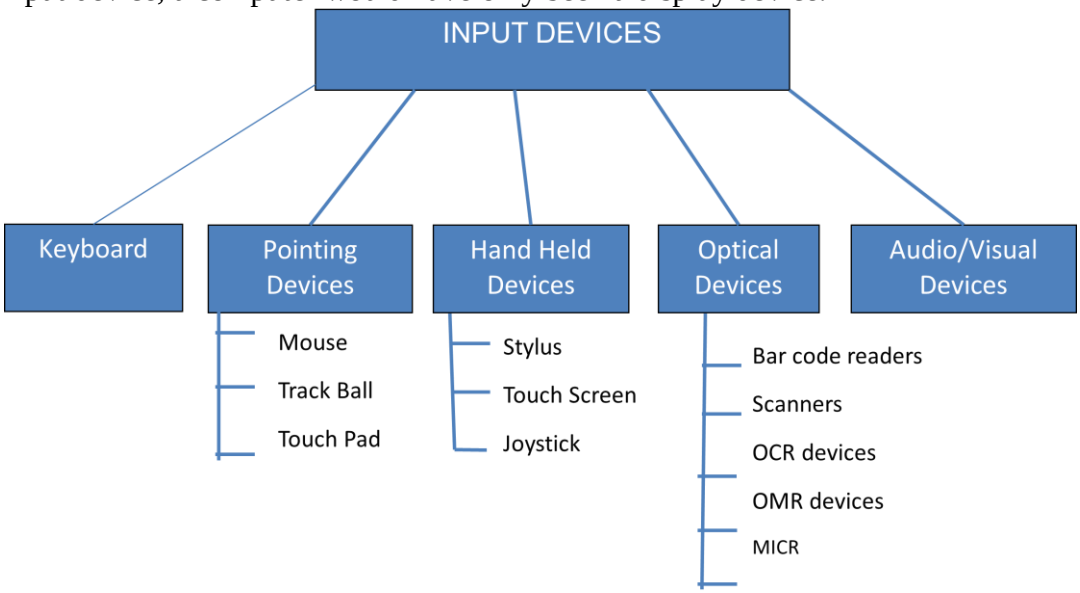
Figure 1.5. Computer's motherboard

Module 1

Input and Output Devices

2.1 Input Devices

- An input device is used to feed data and instructions into the computer. In the absence of an input device, a computer would have only been a display device.

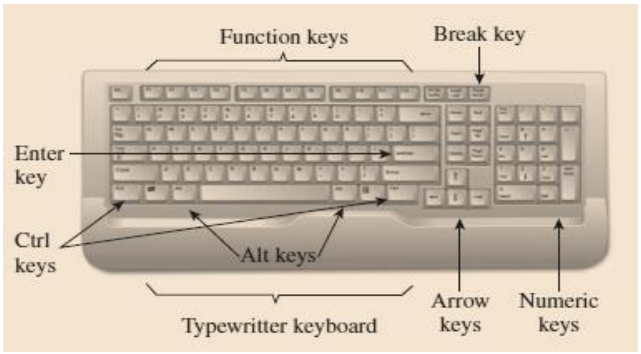


Categories of input devices

2.1.1Keyboard

- With a keyboard, the user can type a document, use keystroke shortcuts, access menus, play games and perform numerous other tasks. Most keyboards have between 80 and 110 keys which include:

Typing keys: These include the letters of the alphabet. The layout of the keyboard is known as **QWERTY** for its first 6 letters.



Numeric keys: These include a set of keys, arranged in the same configuration found on calculators to speed up data entry of numbers.

Function keys: These are used by applications and operating system to input specific commands. They are often placed on the top of the keyboard in a single row.

Control keys: These keys provide cursor and screen control. It includes four directional arrow key. Control keys also include Home, End, Insert, Delete, Page Up, Page Down, Control (Ctrl), Alternate (Alt), Escape(Esc).

Special purpose keys: Keyboard also contains some special purpose keys such as Enter, Shift, Caps Lock, Num Lock, Space bar, Tab, and Print screen.

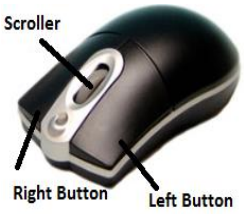
Advantages*:* Easy to use and inexpensive.

Disadvantages*:*

- Keyboard cannot be used to draw figures.
- The process of moving the cursor to some other position is very slow.
- Mouse and other pointing devices are more apt for this purpose

2.1.2 Pointing Devices

1. MOUSE



The mouse is the key input device to be used in a graphical user interface (GUI). The users can use mouse to handle the pointer easily on the screen to perform various functions like opening a program or file.

With mouse, the users no longer need to memorize commands, which was earlier a necessity when working with text-based command line environment such as MS-DOS.

Advantages:

- Easy to use; Cheap; Can be used to quickly place the cursor anywhere on the screen;
- Helps to quickly and easily draw figures;
- Point and click capabilities makes it unnecessary to remember certain commands

Disadvantages*:*

- Needs extra desk space to be placed and moved easily.
- The ball in the mechanical mouse must be cleaned to remove dust from it

2. TRACKBALL

A trackball is a pointing device which is used to control the position of the cursor on the screen. These are usually used in notebook and laptop computers where it is placed on the keyboard. The trackball is nothing but an upside-down mouse that rotates in place within a socket. The user rolls the ball to position the cursor at an appropriate position on the screen and then clicks one of the buttons to select objects or position the cursor for text entry.

Advantages:

- Trackball provides better resolution; Occupies less space
- Easier to use as compared to mouse as its use involves less hands-and-arms movements.

Disadvantage:

The trackball chamber is often covered with dust, so it must be cleaned regularly.

3. TOUCHPAD

A touchpad (or track pad) is a small, flat, rectangular stationary pointing device with sensitive surface of 1.5 or 2 inches square. The user has to slide his finger tips across the surface of the pad to point to a specific object on the screen.

The surface translates the motion and position of user's fingers to a relative position on the screen. There are also buttons around the

edge of the pad that work like mouse buttons. Touchpad is widely used in laptops and is built-in on the keyboard.

Advantages*:*

- Occupies less space.
- Easier to use as compared to mouse as its use involves less hands-and-arms movements.
- It is built-in the keyboard, so no need to carry an extra device separately.

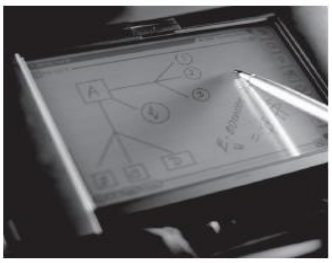
2.1.3 Handheld Devices

- A handheld device is a pocket-sized computing device with a display screen and touch input and/or miniature keyboard.

1. **JOYSTICK:**

It is a cursor control device widely used in computer games and CAD/CAM applications. It consists of a hand-held lever that pivots on one end and transmits its coordinates to a whose position can also be read by the computer.

2. **STYLUS**:



A stylus is a pen-shaped input device used to enter information or write on the touch screen of a phone. Stylus is a small stick that can also be used to draw lines on a surface as input to a computer, choose an option from a menu, move the cursor to another location on the screen, take notes and create short messages. The stylus usually slides into a slot built into the smart phone for that purpose.

3. **TOUCH SCREEN:**

A touch screen is a display screen which can identify the occurrence and position of a touch inside the display region. The user can touch the screen either by his finger or by using a stylus. These displays can be connected to computers, laptops, PDAs, cell phones etc.

2.1.4 Optical Devices

- Optical devices, also known as data-scanning devices, use light as a source of input for detecting or recognizing different objects such as characters, marks, codes and images. These devices convert these objects into digital data and send it to the computer for further processing.

1. BARCODE READERS:

A barcode reader (or price scanner or point-of-sale scanner) is a hand-held input device which is used to capture and read information stored in a barcode. A barcode reader consists of a scanner, a decoder, and a cable used to connect the reader with a computer.

The barcode reader merely captures and translates the barcode into numbers and/or letters. To make use of the information captured it must be connected to a computer for further processing. For this purpose, the barcode reader is connected to a computer through a serial port, keyboard port, or an interface device called a wedge.

Advantages*:*

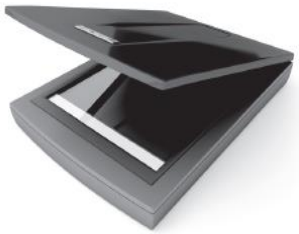
Cheap

Portable

Used to read data stored in bar codes

Handy and easy to use

2. IMAGE SCANNER:

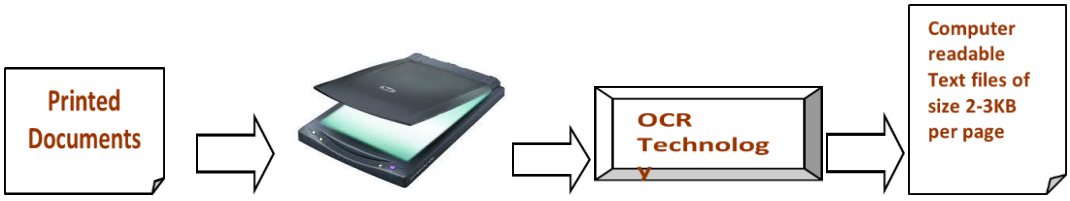


- It is a device that captures images, printed text, handwriting from different sources and converts it into a digital image for computer editing and display.
- Scanners come in hand-held, feed-in, and flatbed types.
- In the flat bed scanner, the object to be scanned is placed on a glass pane and a sensor and light moves along the pane, reflecting off the image placed on the glass.
- A hand image scanner has to be manually moved across an object or image to be scanned. The scanner produces light from green LEDs which highlight and scan the image onto a computer for further processing.
- Film scanners are usually used in photography and slides. The slide or negative film is first inserted in strips of six or less frames into the film scanner, and then moved across a lens and censor to capture the image.

3. Optical Character Recognition (OCR):

OCR is the process of converting printed materials into text or word processing files that can be easily edited and stored. The steps involved in OCR include:

- Scanning of the text character-by-character.
- Analyzing the scanned-in image to translate the character image into character codes (e.g. ASCII).



Scans the documents and makes a bitmap of size 50-150KB per page

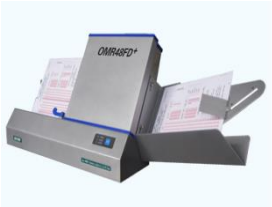
Advantages:

- Printed documents can be converted into text files.
- Advanced OCR can recognize handwritten text and convert them into computer readable text files.

Disadvantages:

- OCR cannot recognize all types of fonts.
- Documents that are poorly types or have strikeover cannot be recognized.
- Very old documents when passed through OCR may not have an exact copy of the text file.

4. OPTICAL MARK RECOGNITION:



OMR is the process of electronically extracting data from marked fields, such as checkboxes and fill-infields, on printed forms. The optical mark reader is fed with an OMR sheet to detect the presence of a mark by measuring reflected light levels. The OM reader interprets this pattern marks and spaces and stores the interpreted data in computer for storage, analysis and reporting.

The error rate for OMR technology is less than 1%.

It is used for applications in which large numbers of hand-filled forms have to be quickly processed with great accuracy, such as surveys, reply cards, questionnaires, ballots or sheets for multiple choice questions.

Advantages*:*

- Optical mark readers works with a very fast speed. They can read up to 9,000 forms per hour .
- They are accurate machines with error rates of just 1% .

5. MAGNETIC INK CHARACTER READER (MICR):

MICR is used to verify the legitimacy or originality of paper documents, especially checks.

MICR consists of magnetic ink printed characters which can be recognized by high speed magnetic recognition devices.

The printed characters provides important information (like check number, bank routing number, checking account number and in some cases the amount of the check) for processing to the receiving party.

MICR is widely used to enhance security, speed up the sorting of documents and minimize the exposure to check fraud.

2.1.5 Audiovisual Input Devices

- **Audio devices** are used to either capture or create sound. They enable computers to accept music, speech or sound affects for recording and/or editing. Microphone and CD player are examples of two widely used audio input devices. Microphone feeds audio input to the computer. The computer must have a sound card to convert analog signals generated through microphone into digital data so that it can be stored in the computer. When the user wants to

hear the pre-recorded audio input, the sound card converts the digital data into analog signals and sends it to the speakers.

Advantages:

- Audio devices can be used by people who have visual problems.
- It is best used in situations where users want to avoid i/p through keyboard or mouse.

Disadvantages:

- Audio input devices are not suitable in noisy places.
- With audio input devices it is difficult to clearly distinguish between two similar sounding words like "sea" and "see".

Video Input Devices:

Video input devices are used to capture video from the outside world into the computer.

Digital camera is used to capture images or videos. It digitizes the image or video and stores them on a memory card. The data can then be transferred to the computer using a cable which connects computer to the digital camera.

W*eb cameras* also capture videos which can be transferred via internet in real time.

Advantages:

- Video input devices are very useful for applications like video conferencing.
- Video input devices can be used to record memorable moments in one's life.
- Video input devices can be used to check security.

Disadvantages*:*

- Videos and images captured using video input devices have a very big file size and there must be compressed before being stored on the computer.

2.1.6 Output Devices

- Any device that outputs/gives information from a computer is called an output device.

Output devices are electromechanical devices which accept digital data from the computer and converts them into human understandable language.

SOFT COPY DEVICES:

Soft copy output devices are those output devices which produce an electronic version of an output. For example, a file which is stored on hard disk, CD, pen drive, etc and is displayed on the computer screen (monitor).

Features of a soft copy output include:

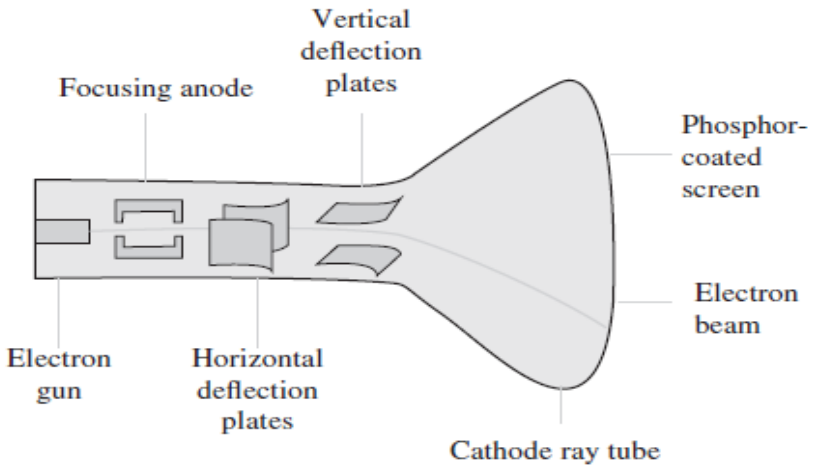
- The output can be viewed only when the computer is on.
- The user can easily edit the soft copy output.
- Soft copy cannot be used by people who do not have a computer.
- Searching data in a soft copy is easy and fast.
- Electronic distribution of a soft copy is cheaper. It can be done easily and quickly.

1. MONITORS:

The Monitor is a soft copy output device used to display video and graphics information generated by the computer through the video card.

A. CATHODE RAY TUBE MONITORS

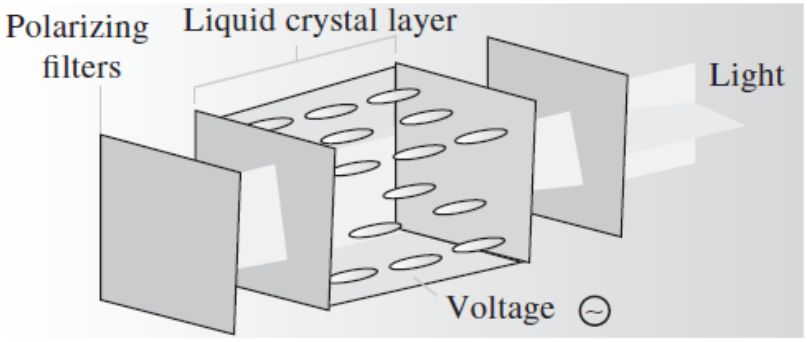
- CRT monitors work by firing charged electrons at a phosphorus film. When electron hit the phosphor coated screen, it glows thereby enabling the user to see the output.
- In a cathode ray tube, the "cathode" is a heated filament which is placed in a vacuum created inside a glass "tube." The "ray" is a stream of electrons which comes out from a heated cathode into the vacuum.
- The focusing anode focuses the stream of electrons to form a tight beam which is then accelerated by an accelerating anode.
- This tight, high-speed beam of electrons flies through the vacuum in the tube and hits the flat screen at the other end of the tube.
- This screen is coated with phosphor, which glows when struck by the beam, thereby displaying the picture which the user sees on the monitor.



Schematic diagram of a Cathode Ray Tube

B. LIQUID CRYSTAL DISPLAY MONITORS

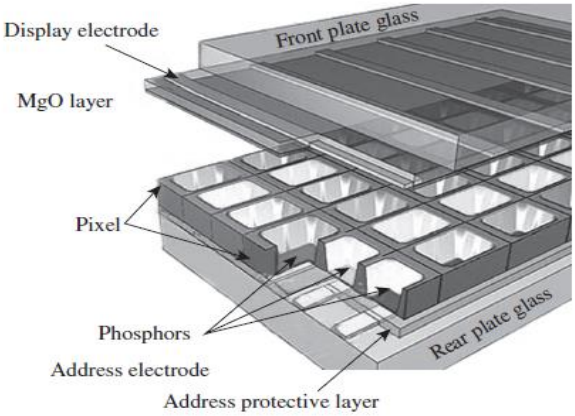
LCD monitor is a thin, flat electronic visual display that uses the light modulating properties of liquid crystals which do not emit light directly. LCD screens are used in a wide range of applications ranging from computer monitors, television, instrument panels, aircraft cockpit displays, signage, to consumer devices like such as video players, gaming devices, clocks, watches, calculators, and telephones. Liquid crystal display technology is based on blocking light. The LCD consists of two pieces of polarizing filters (or substrates) that contain a liquid crystal material between them. A backlight creates light which is made to pass through the first substrate. Simultaneously, the electrical currents cause the liquid crystal molecules to align to allow varying levels of light to pass through to the second substrate and create the colors and images are seen on the screen.



Schematic diagram of a Liquid Crystal Display Monitor

C. PLASMA MONITORS:

- Plasma monitors are thin and flat monitors widely used in TVs and computers. The plasma display contains two glass plates that have tiny cells filled with xenon and neon gas.
- The display electrode is covered by a magnesium oxide protective layer and is arranged in horizontal rows along the screen while the address electrodes are arranged in vertical columns thereby forming grid like structure.
- To ionize the gas in a particular cell, the electrodes that intersect at that cell are charged at least thousands of times in a small fraction of a second.
- An electric current begins to flow through the gas in the cell. The current creates a rapid flow of charged particles thereby stimulating the gas atoms to release ultraviolet photons.
- When these UV photons hit a phosphor atom in the cell, one of the phosphor's electrons jumps to a higher energy level and the atom heats up. When the electron falls back to its normal level, it releases energy in the form of a visible light photon.



Schematic diagram of a Plasma Monitor

2. PROJECTOR:

A projector is a device which takes an image from a video source and projects it onto a screen or other surface. These days, projectors are used for a wide range of applications varying from home theater e systems to organizations for projecting information and presentations onto screens large enough for rooms filled with people to see.

3. SPEAKERS :

Today all business and home users demand sound capabilities and thus different types of speakers to enable users to enjoy music, movie, or a game and the voice will be spread through the entire room. With good quality speakers, the voice will also be audible even to people sitting in another or room or even to neighbors.

However, in case the user wants to enjoy loud music without disturbing the people around him, he can use a headphone.

Another device called headset was developed to allow the users to talk and listen at the same time, using the same device.

(a)Speakers

(b)headphone

(c)headset

HARD COPY OUTPUT DEVICES:

Hard copy output devices produces a physical form of output. For example, the content of a file printed on a paper is a form of hard copy output.

PRINTERS

Printer is a device that outputs text and graphics information obtained from the computer and prints it on to a paper. Printers are available in the market in a variety of size, speed, sophistication, and cost. The qualities of printer which are of interest to users include:

Color*:* Colored printouts are needed for presentations or maps and other pages where color is part of the information. They are more expensive.

Memory*:* Most printers have a small amount of memory that can be expanded by the user. Having more memory makes enhances the speed of printing

Resolution*:* The resolution of a printer means the sharpness of text and images on paper. It is usually expressed in dots per inch (dpi). Even the least inexpensive printer provides sufficient resolution for most purposes at 600 dpi.

Speed: Speed means number of pages that are printed in one minute. While high speed printers are a little expensive, the inexpensive printers on the other hand can print only about 3 to 6 sheets per minute. Color printing is even slower.

A. Impact Printer: They create characters by striking an inked ribbon against the paper. Example: dot-matrix printers, daisywheel printers, and most types of line printer.

B. Non Impact Printer: Non-impact printers are much quieter than impact printers as their printing heads do not strike the paper. They offer better print quality, faster printing and the ability to create prints that contain sophisticated graphics.

Non-impact printers use either solid or liquid cartridge-based ink which is either sprayed, dripped or electro statically drawn onto the page. The main types of non-impact printer are: inkjet, printer, laser printer and thermal printer.

DOT MATRIX PRINTER

A dot matrix printer prints characters and images of all types as a pattern of dots. It has a print head (or hammer) that consists of pins representing the character or image. The print head runs back and forth, or in an up and down motion, on the page and prints by striking an ink-soaked cloth ribbon against the paper, much like the print mechanism on a typewriter.

Advantages:

- It can produce carbon copies; offers lowest printing cost per page.
- widely used for bulk printing where quality of the print is not of much importance; is cheap; When the ink is about to finish,
- The printout gradually fades rather than suddenly stopping partway through a job.
- It can use continuous paper rather than individual sheets, making them useful for data logging.

Disadvantages :

- It creates a lot of noise when the pins strike the ribbon to the paper.
- It can only print lower-resolution graphics, with limited quality.
- It is very slow.
- Poor print quality.

DAISY WHEEL PRINTER

- Daisy wheel printers use an impact printing technology to generate high-quality output comparable to typewriters but three times faster.
- The print head of a daisy wheel printer is a circular wheel, about 3 inches in diameter with arms or spokes. The characters are embossed at the outer end of the arms.
- To print a character, the wheel is rotated in such a way that the character to be printed is positioned just in front of the printer ribbon.

- The spoke containing the required character is then hit by a hammer thereby striking the ribbon to leave an impression on the paper placed behind the ribbon. Movement of all these parts is controlled by microprocessor in the printer.
- The key benefit of using a daisy wheel printer is that the print quality is high as the exact shape of the character hits the ribbon to leave and impression on paper.

LINE PRINTER:

Line printer is a high speed impact printer in which one typed line is printed at a time. The speed of a line printer usually varies from 600 to 1200 lines-per-minute or approximately 10 to 20 pages per minute. They are widely used in datacenters and in industrial environments. Band printer is a commonly used variant of line printers.

Band Printer: A band printer (loop printer) is an impact printer. The set of characters are permanently embossed on the band and this set cannot be changed unless the band is replaced. The band itself revolves around hammers that push the paper against the ribbon, allowing the desired character to be produced on the paper. However, band printers cannot be used for any graphics printing as the characters are predetermined and cannot be changed unless the band is changed.

INKJET PRINTERS

- In inkjet printers, the print head has several tiny nozzles, also called jets.
- As the paper moves past the print head, the nozzles spray ink onto it, forming the characters and images.
- The dots are extremely small (usually between 50 and 60 microns in diameter) and are positioned very precisely, with resolutions of up to 1440x720 dots per inch (dpi).
- There is usually one black ink cartridge and one so-called color cartridge containing ink in primary pigments (cyan, magenta, and yellow).
- While inkjet printers are cheaper than laser printers, they are more expensive to maintain. The cartridges of inkjet printers have to be changed more frequently and the special coated paper required to produce high-quality output is very expensive. So the cost per page of inkjet printers becomes ten times more expensive than laser printers. Therefore, inkjet printers are not well-suited for high-volume print jobs.

LASER PRINTER

- It is a non-impact printer that works at a very high speed and produces high quality text and graphics.
- It uses the photocopier technology. When a document is sent to the printer, a laser beam "draws" the document on a drum (which is coated with a photo-conductive material) using electrical charges.
- After the drum is charged, it is rolled in toner (a dry powder type of ink).
- The toner sticks to the charged image on the drum.

- The toner is transferred onto a piece of paper and fused to the paper with heat and pressure.
- After the document is printed, the electrical charge is removed from the drum and the excess toner is collected.
- While color laser printers are also available in the market but users prefer only monochrome printers because a color laser printer is up to 10 times more expensive than a monochrome laser printer.

PLOTTERS

A plotter is used to print vector graphics with a high print quality. They are widely used to draw maps, in scientific applications and in applications like CAD, CAM and CAE.

- A **drum plotter** is used to draw graphics on a paper that is wrapped around a drum. It works by rotating the drum back and forth to produce vertical motion. The pen which is mounted on a carriage is moved across the width of the paper. Hence, the vertical movement of the paper and the horizontal movement of the pen create the required design under the control of the computer.
- In a **flatbed plotter**, the paper is spread on the flat rectangular surface of the plotter and the pen is moved over it. Flatbed plotters are less expensive and used in many smaller computing systems. In this type of plotter, the paper is not moved rather plotting is done by moving an arm that moves a pen over paper.

Module-1

Designing Efficient Programs

3.1 Programming Paradigms

- ✓ **A programming paradigm is a fundamental style of programming that defines how the structure and basic elements of a computer program will be built.**
- ✓ The style of writing programs and the set of capabilities and limitations that a particular programming language has depends on the programming paradigm it supports.
- ✓ These paradigms, in sequence of their application, can be classified as follows:
  - ➢ **Monolithic programming** — emphasizes on finding a solution.
  - ➢ **Procedural programming**—lays stress on algorithms.
  - ➢ **Structured programming**—focuses on modules.
  - ➢ **Object-oriented programming**—emphasizes on classes and objects.
  - ➢ **Logic-oriented programming**—focuses on goals usually expressed in predicate calculus.
  - ➢ **Rule-oriented programming**—makes use of 'if-then-else' rules for computation.
  - ➢ **Constraint-oriented programming**—utilizes invariant relationships to solve a problem.

1. Monolithic Programming

- ✓ Monolithic programs have just **one program module.**
- ✓ Such programming languages **do not support the concept of subroutines.**
- ✓ Therefore, all the actions required to complete a particular task are embedded within the same application itself. This not only makes the **size of the program large** but also makes it **difficult to debug and maintain.**
- ✓ Programs written using monolithic programming languages consist of **global data and sequential code.**
- ✓ **Sequential code** is one in which **all instructions are executed in the specified sequence.**
- ✓ In order to change the sequence of instructions, **jump statements or 'goto' statements** are used.
- ✓ The **global data** can be **accessed and modified from any part of the program,** thereby posing a serious threat to its integrity (no security for data).
- ✓ **Examples** are **Assembly Language and BASIC**.

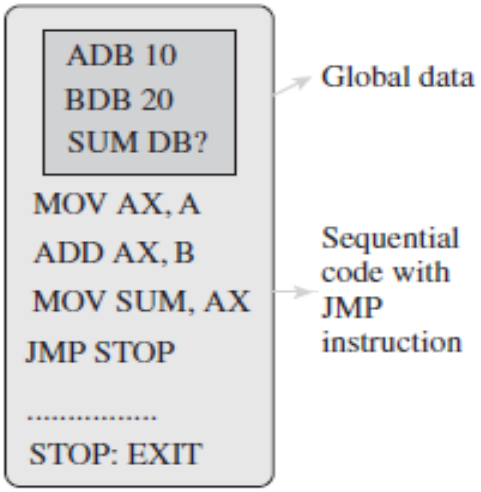


Figure 3.1. Structure of a monolithic program

2. Procedural Programming

- ✓ In procedural languages, **a program is divided into subroutines that can access global data.**
- ✓ To **avoid repetition of code**, each subroutine performs a **well-defined task**.
- ✓ A subroutine that needs the service provided by another subroutine can call that subroutine. Therefore, with '**jump', 'goto', and 'call' instructions**, the sequence of execution of instructions can be altered.
- ✓ **Examples FORTRAN and COBOL** are two popular procedural programming languages

Advantages:

- ✓ The only goal is to write **correct programs.**
- ✓ Programs are **easier to write** as compared to monolithic programming.

Disadvantages:

- ✓ **No concept of reusability.**
- ✓ Requires **more time and effort** to write programs.
- ✓ Programs are **difficult to maintain**.
- ✓ **Global data is shared** and therefore may get altered (mistakenly).

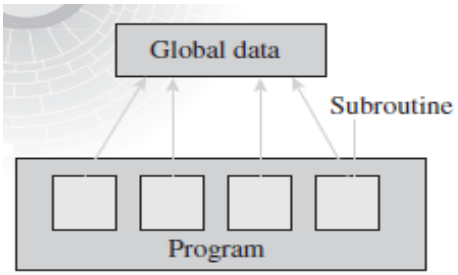


Figure 3.2. Structure of a procedural program

3. Structured Programming

- ✓ Structured programming employs a **top-down approach in which the overall program structure is broken down into separate modules.**
- ✓ This allows the **code to be loaded into memory more efficiently and also be reused** in other programs.
- ✓ **Modules are coded separately** and once a module is written and tested individually, it is then **integrated with other modules** to form the overall program.
- ✓ Structured programming is based on modularization which groups related statements together into modules.

Advantages:

- ✓ The goal of structured programming is to write **correct programs** that are **easy to understand and change.**
- ✓ Modules enhance **programmers productivity** by allowing them to look at the big picture first and focus on details later.
- ✓ With modules, many programmers can work on a single, large program, with each working on a different module.
- ✓ A structured program **takes less time to be written** than other programs.
- ✓ Each **module** has its own **local data.**
- ✓ A structured program is **easy to debug** because each procedure is specialized to perform just one task and every procedure can be checked individually for the presence of any error.
- ✓ Individual procedures are **easy to change as well as understand.**

- ✓ More **emphasis is given on the code** and the **least importance is given to the data.**
- ✓ **Modules or procedures** written for one program can be **reused in other programs** as well.

Disadvantages:

- ✓ **Not data-centred.**
- ✓ **Global data is shared** and therefore may get **inadvertently modified.**
- ✓ **Main focus** is on **functions.**

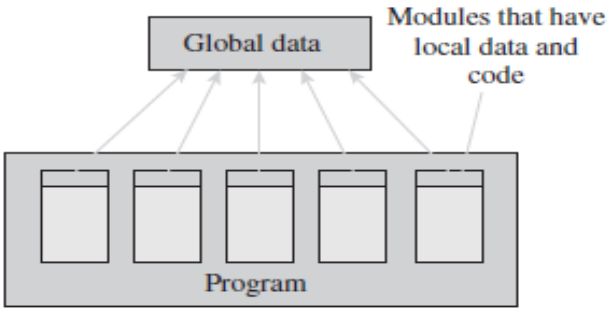


Figure 3.3. Structured program

4. Object-Oriented Programming (OOP)

- ✓ The object-oriented paradigm is **task-based and data-based.**
- ✓ In this paradigm, all the **relevant data and tasks** are grouped together in entities known as **objects.**
- ✓ It treats **data as a critical element** in the program development and **restricts its flow freely around the system.**
- ✓ For example, consider a list of numbers. In the object-oriented paradigm, the **list and the associated operations** are treated as one entity known as an object. In this approach, the list is considered an object consisting of the list, along with a collection of routines for manipulating the list.
- ✓ The striking features of OOP include the following:
    - Programs are **data centred.**
    - Programs are divided in terms of **objects** and not procedures.
    - **Functions** that operate on data are **tied together with the data.**
    - **Data is hidden** and not accessible by external functions.
    - **New data and functions** can be **easily added** as and when required.
    - Follows a **bottom-up approach** for problem solving.

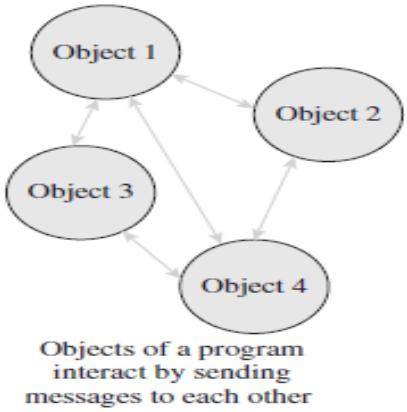


Figure 3.4. Object-oriented paradigm

3.2 Design and Implementation of Efficient Programs

- ✓ To design and develop **correct, efficient, and maintainable** programs, the entire program development process is divided into a **number of phases** where each phase performs a **well-defined task.**
- ✓ Output of one phase provides the input for its subsequent phase.
- ✓ The phases in the software development life cycle (SDLC) process is shown in Figure 3.5.

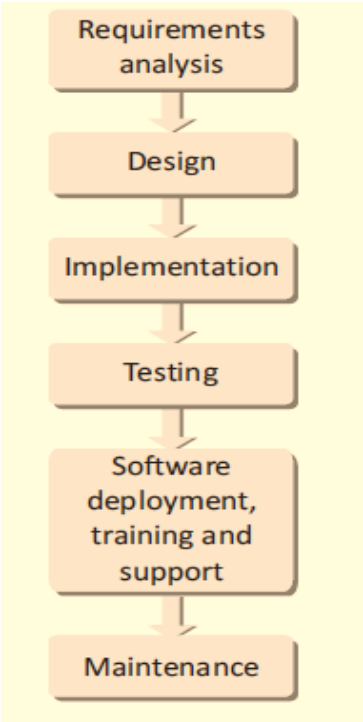


Figure 3.5. Phases in Software Development Life Cycle (SDLC)

1. Requirements Analysis

- ✓ In this phase, **user's expectations** are gathered to know why the **program/software has to be built.**
- ✓ All the gathered requirements are **analysed** gathered the **scope** to arrive at or the **objective of the overall software product.**
- ✓ All the gathered requirements are **documented** to avoid any doubts or uncertainty regarding the functionality of the programs.

2. Design

- ✓ A **plan of actions** is made before the actual development process could start.
- ✓ The core structure of the software/program is broken down into **modules.**
- ✓ The solution of the program is then specified for each module in the form of **algorithms, flowcharts, or pseudocodes.**

3. Implementation

- ✓ **Designed algorithms are converted into program code** using any of the **high level languages.**
- ✓ The choice of language depends on the type of program like whether it is a **system or an application program.**
- ✓ **Program codes are tested** by the programmer to ensure their **correctness.**
- ✓ While constructing the code, the **development team** checks whether the **software is compatible** with the **available hardware and other software components** that were mentioned in the Requirements Specification Document created in the first phase.

4. Testing

- ✓ **All the modules are tested together** to ensure that the **overall system works well as a whole product.**
- ✓ Although individual pieces of codes are already tested by the programmers in the implementation phase, there is always **a chance for bugs to creep in the program when the individual modules are integrated** to form the overall program structure.
- ✓ Software is **tested using a large number of varied inputs** also known as **test data** to ensure that the software is working as expected by the users' requirements that were identified in the requirements analysis phase.

5. Software Deployment, Training and Support:

- ✓ After testing the **software is deployed in the production environment.**
- ✓ Software **Training and Support** is a crucial phase which makes the **end users familiar with how to use the software.**
- ✓ Moreover, people are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it has become very **crucial to have training classes** for the users of the software.

6. Maintenance

- ✓ Maintenance and enhancements are ongoing activities which are done to **cope with newly discovered problems or new requirements.**
- ✓ Such activities may take a **long time to complete** as the requirement may call for addition of new code that does not fit the original design or an extra piece of code required to fix an unforeseen problem.

3.3 Program Design Tools: Algorithms, Flowcharts, Pseudocodes

3.3.1 Algorithms

- ✓ **An algorithm provides a blueprint to writing a program to solve a particular problem.**
  or
  The algorithm is a 'step-by-step procedure' to be followed in solving a problem.
- ✓ It is considered to be an effective procedure for solving a problem in a **finite number of steps.**
- ✓ A well-defined algorithm **always provides an answer**, and is guaranteed to **terminate.**
- ✓ Algorithms are mainly used to **achieve software re-use.**
- ✓ A good algorithm must have the following characteristics:
  - **Be precise.**
  - **Be unambiguous.**
  - **Not even a single instruction must be repeated infinitely.**
  - **After the algorithm gets terminated, the desired result must be obtained.**

Control Structures Used In Algorithms

1. Sequence: Sequence means that **each step of the algorithm is executed in the specified order.**

Example: Algorithm to Add two numbers

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: SET SUM = A + B
Step 4: PRINT SUM
Step 5: END
```

2. Decision: Decision statements are used when **the outcome of the process depends on some condition.** For example, if x=y, then print "EQUAL". Hence, the general form of the if construct can be given as:
if condition then process.

Example: Algorithm to test Equality of two number

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: IF A = B
            Then PRINT "EQUAL"
        ELSE
             PRINT "NOT EQUAL"
Step 4: END
```

3. Repetition: Repetition, which involves **executing one or more steps for a number of times**, can be implemented using constructs such as **while, do-while, and for loops**. These loops execute one or more steps **until some condition is true.**

Example: Algorithm to print first 10 Natural numbers

```
Step 1: [INITIALIZE] SET I = 0, N = 10
Step 2: Repeat Step while I<=N
Step 3:   PRINT I
Step 4: SET I + 1
Step 5: END
```

Examples:

1. Write an algorithm to find the larger of two numbers.

Solution

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A > B
           Print A
        ELSE IF A < B
           Print B
        ELSE
           Print "The numbers are equal"
        [END OF IF]
Step 4: End
```

2. Write an algorithm to find whether a number is even or odd.

Solution

```
Step 1: Input number as A
Step 2: IF A % 2 = 0
           Print "Even"
        ELSE
           Print "Odd"
        [END OF IF]
Step 3: End
```

3. Write an Algorithm to Swap two numbers.

Solution

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: Set temp = A
Step 4: Set A = B
Step 5: Set B = temp
Step 6: Print A, B
Step 7: End
```

4. Write an algorithm to print the grade obtained by a student using the following rules:

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

Solution

```
Step 1: Enter the marks obtained as M
Step 2: IF M > 75
          Print "O"
Step 3: IF M >= 60 and M < 75
          Print "A"
Step 4: IF M >= 50 and M < 60
          Print "B"
Step 5: IF M >= 40 and M < 50
          Print "C"
        ELSE
          Print "D"
        [END OF IF]
Step 6: End
```

5. Write an algorithm to compute the simple interest.

Solution:

Step 1: [Initialize] Start
Step 2: [Input the values of P, T, R]
Read P, T, R
Step 3: [Compute the Simple Interest]
SI= (P*T*R)/100
Step 4: [Display the Simple Interest]
Print SI
Step 5: [Finished] Stop

6. Write an algorithm to compute the area and perimeter of a circle.
Solution:

Step 1: Start
Step 2: Read r
Step 3: a= 3.142*r*r
p= 2*3.142*r
Step 4: Print a,p
Step 5: Stop

7. Write an algorithm to compute the area and perimeter of a rectangle.
Solution:

Step 1: Start
Step 2: Read l,b
Step 3: a= l*b
p= 2*(l+b)
Step 4: Print a, p
Step 5: Stop

8. Write an algorithm to compute the perimeter and area of a triangle when three sides are given.
Solution:

Step 1: Start
Step 2: Read a, b, c
Step 3: s= (a + b + c)/2
a= sqrt(s*(s-a)*(s-b)*(s-c))
p= a + b + c
Step 4: Print a, p
Step 5: Stop

9. Write an algorithm to compute the compound interest.
Solution:

Step 1: Start
Step 2: Read P, T, R
Step 3:CI= $P*(1+R/100)^{T}-P$
Step 4:Print CI
Step 5: Stop

10. Write an algorithm to simulate simple calculator or to perform arithmetic operations.
Solution:

Step 1: Start
Step 2: Read n1, n2
Step 3: sum = n1 + n2
sub = n1 - n2
mul = n1 * n2
div = n1 / n2
Step 4: Print sum, sub, mul, div
Step 5: Stop

11. Write an algorithm to calculate the area and volume of sphere.
Solution:

Step 1: Start
Step 2: Read r
Step 3: A= 4*3.142*r*r
V= (4/3)*3.142*r*r*r
Step 4: Print A, V
Step 5: Stop

12. Write an algorithm to calculate the area and volume of cube.
Solution:

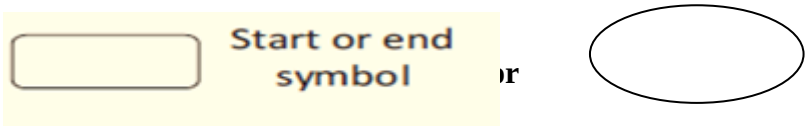
Step 1: Start
Step 2: Read s
Step 3: A=6*s*s
V= s*s*s
Step 4: Print A, V
Step 5: Stop

13. Write an algorithm to convert degrees in Fahrenheit to degrees in Celsius.
Solution:

Step 1: Start
Step 2: Read f
Step 3: c=(5/9)*(f-32)
Step 4: Print c
Step 5: Stop

3.3.2 Flowcharts

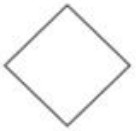
- ✓ **Flowchart is a graphical or symbolic representation of a process.**
  or
- ✓ **A flowchart is a "graphical or symbolic" representation of an algorithm.**
- ✓ It is basically used to **design and document virtually complex processes** to help the **viewers to visualize the logic of the process**, so that they can gain a **better understanding of the process and find flaws, bottlenecks, and other less obvious features within it.**
- ✓ When designing a flowchart, each step in the **process is depicted by a different symbol and is associated with a short description.** The symbols in the flowchart are **linked together with arrows** to show the flow of logic in the process.



or

- Rectangle- Computational steps (used while assigning and calculating some results)

- Parallelogram- Reading operation used for Input / Output or data or Information from/to any device



- Diamond- Decision (questions) and consequently the branch points

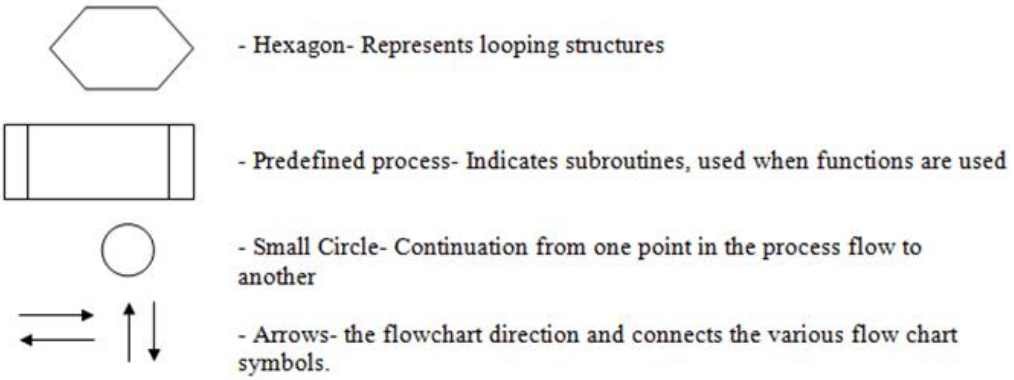


Figure 3.6. Symbols of flowchart

Advantages of Flowcharts:

- ✓ They act as a **guide or blueprint for the programmer** to code the solution in any programming language.
- ✓ It helps programmers to **understand the logic of complicated and lengthy problems.**
- ✓ They are very **good communication tools** to explain the logic of a system to all concerned.
- ✓ They help to **analyze the problem** in a **more effective manner.**
- ✓ Flowchart can be used to **debug programs** that have error(s).

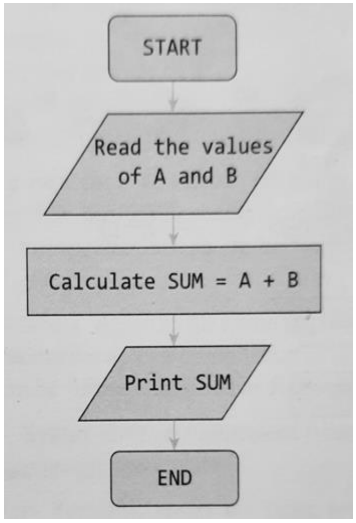
Limitations of Flowcharts:

- ✓ Drawing flowcharts is a **laborious and time consuming** activity.
- ✓ Flowchart of a **complex program becomes complex and clumsy.**
- ✓ A **little bit of alteration** in the solution may require **complete redrawing** of the flowchart.
- ✓ There are **no well-defined standards** that limit the details that must be incorporated in a flow chart.

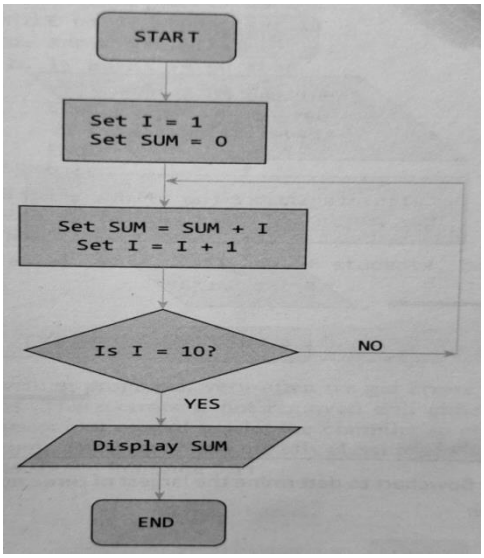
Examples:

1. Draw a flowchart to add two numbers.

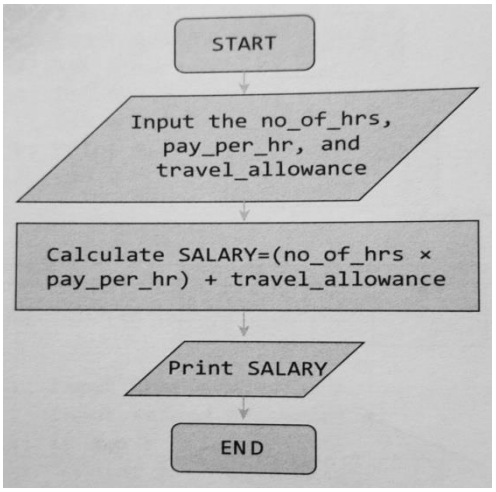
Solution:



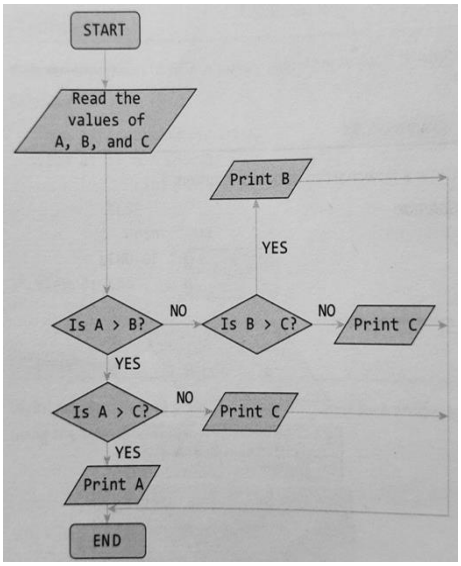
2. Draw a Flowchart to display sum of first 10 natural numbers.
Solution:



3. Draw a Flowchart to calculate the salary of a wager.
Solution:

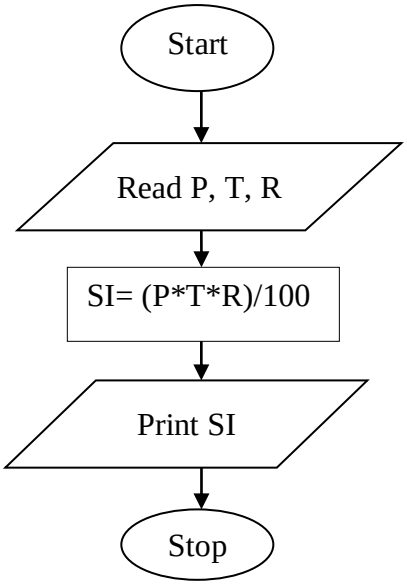


4. Draw a Flowchart to determine the largest of three numbers.
Solution:



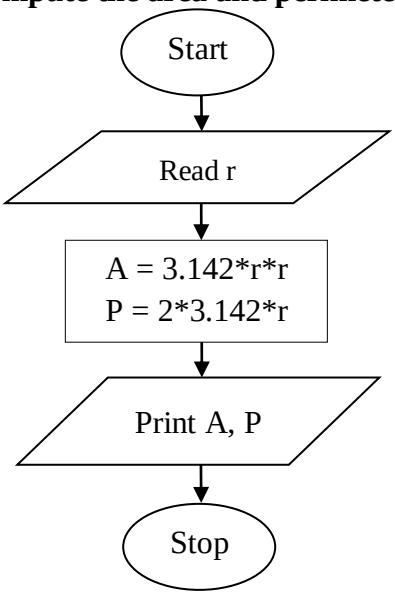
5. Draw the flowchart to compute Simple Interest.

Solution:



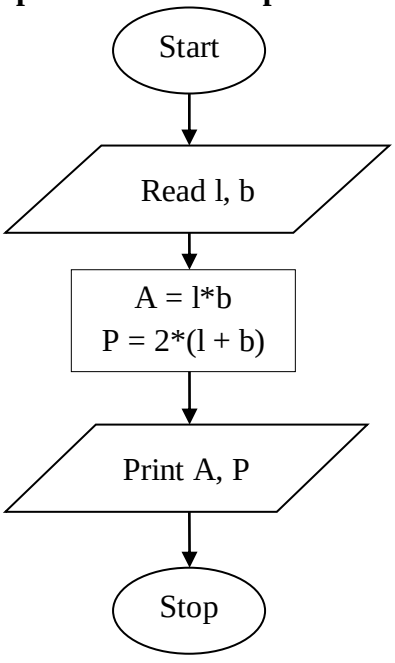
6. Draw the flowchart to compute the area and perimeter of a circle.

Solution:



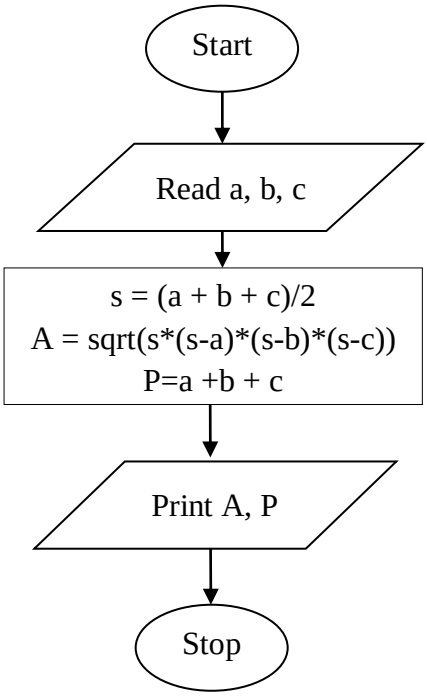
7. Draw the flowchart to compute the area and perimeter of a rectangle.

Solution:



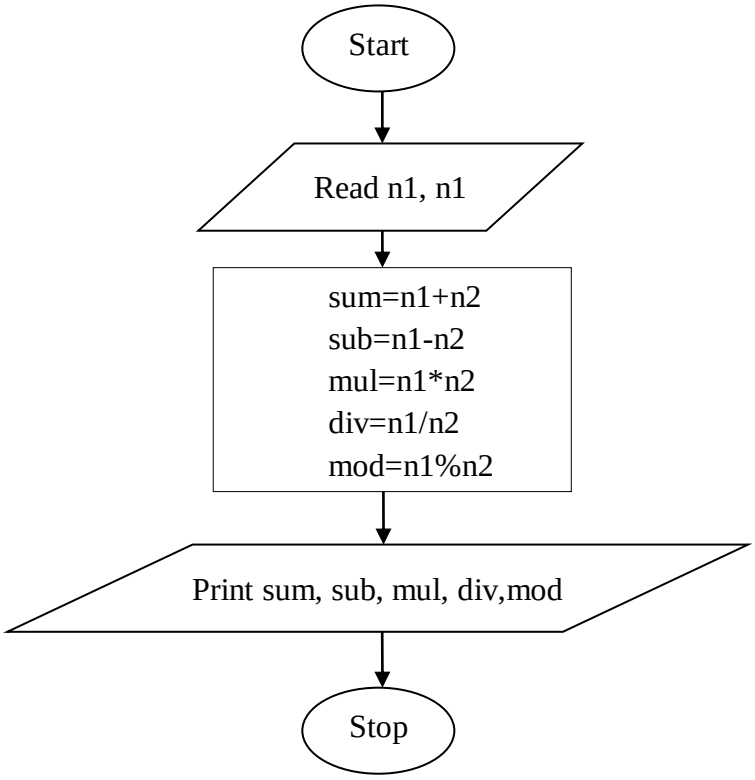
8. Draw the flowchart to compute the area and perimeter of a triangle when three sides are given.

Solution:



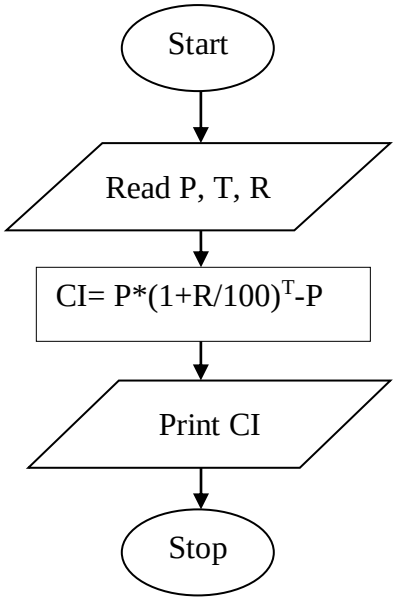
9. Draw the flowchart to simulate the simple calculator.

Solution:



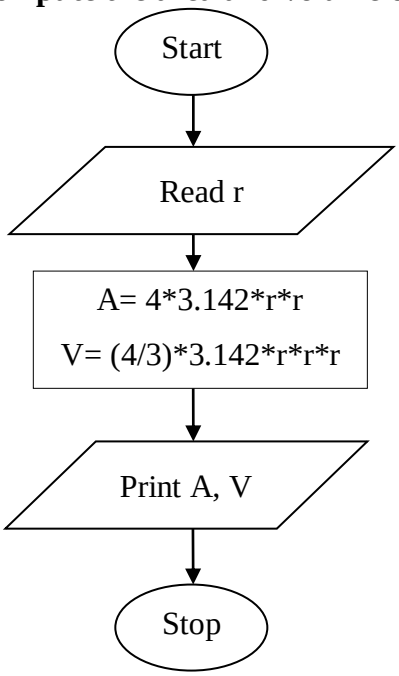
10. Draw the flowchart to compute Compound Interest.

Solution:



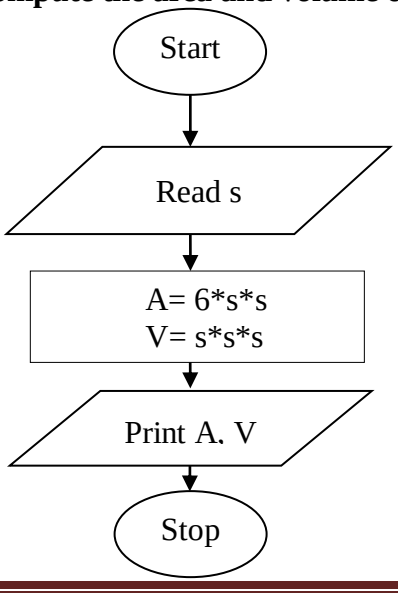
11. Draw the flowchart to compute the area and volume of sphere.

Solution:

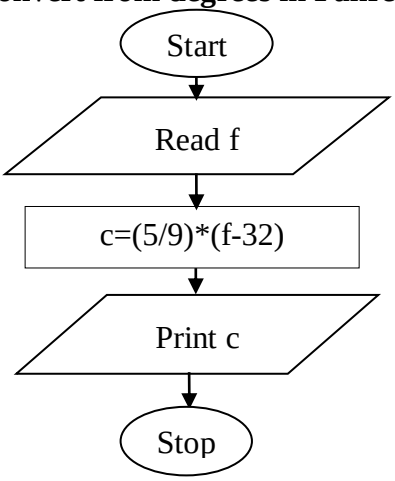


12. Draw the flowchart to compute the area and volume of cube.

Solution:



13. Draw the flowchart to convert from degrees in Fahrenheit to degrees in Celsius.
Solution:



3.3.3 Pseudocode

- ✓ **Pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language.**
  or
  "Pseudo code is nothing but a series of steps to solve a given problem written using a mixture of English language and C like language."
- ✓ Pseudo code consists of statements which are a combination of English and C.
- ✓ It is not quite 'C' code but can be translated.
- ✓ Pseudo code is an outline of a program that can easily be converted into programming language.
- ✓ Flowcharts can be considered as graphical alternatives to pseudo codes but requires more space on paper.

Examples:
1. Write a pseudocode for calculating the price of a product after adding the sales tax to its original price.
Solution:

```
1. Read the price of the product
2. Read the sales tax rate
3. Calculate sales tax = price of the item ×;
   sales tax rate
4. Calculate total price = price of the product
   + sales tax
5. Print total price
6. End
Variables: price of the item, sales tax rate,
   sales tax, total price
```

2. Write a pseudocode to calculate the weekly wages of an employee. The pay depends on wages per hour and the number of hours worked. Moreover, if the employee has worked for more than 30 hours, then he or she gets twice the wages per hour, for every extra hour he or she has worked.
Solution:

```
1. Read hours worked
2. Read wages per hour
3. Set overtime charges to 0
4. Set overtime hrs to 0
5. IF hours worked > 30 then
   a. Calculate overtime hrs = hours worked -
      30
   b. Calculate overtime charges = overtime
      hrs × (2 × wages per hour)
   c. Set hours worked = hours worked -
      overtime hrs
   ENDIF
6. Calculate salary = (hours worked × wages
   per hour) + overtime charges
7. Display salary
5. End
Variables: hours worked, wages per hour,
   overtime charges, overtime hrs, salary
```

3. Write a Pseudo code for printing the number and its square starting from 4 to 9.
Solution:

Begin
Input 4
square=4*4
Print 4,16
do the same for each of the other numbers from 5 to 9
End

4. Write a Pseudo code to compute the Simple Interest.
Solution:

Begin
Input P, T, R
SI= (P*T*R)/100
Print SI
End

5. Write a Pseudo code to compute the Compound Interest.
Solution:

Begin
Input P, T, R
CI= $P*(1+R/100)^{T}-P$
Print CI
End

6. Write a Pseudo code to calculate area and perimeter of circle.
Solution:

Begin
Input r
A=3.142*r*r
P=2*3.142*r
Print A, P
End

7. Write a Pseudo code to calculate the area and perimeter of rectangle.
Solution:

```
Begin
Input l, b
 A=l*b
 P=2*(l+b)
Print A, P
End
```

8. Write a Pseudo code to calculate the area and perimeter of triangle when three sides are given.
Solution:

```
Begin
Input a,b,c
 s= (a + b + c)/2
 A=sqrt(s*(s-a)*(s-b)*(s-c))
 P= a + b +c
Print A, P
End
```

9. Write a Pseudo code to calculate the area and perimeter of sphere.
Solution:

```
Begin
Input r
A= 4*3.142*r*r
 V= (4/3)*3.142*r*r*r
Print A, V
End
```

10. Write a Pseudo code to calculate the area and perimeter of cube.
Solution:

```
Begin
Input s
A= 6*s*s
V= s*s*s
Print A, V
End
```

11. Write a Pseudo code to convert from degrees in Fahrenheit to degrees in Celsius.
Solution:

```
Begin
Input f
c=0.56*(f-32)
 Print c
End
```

3.4 Types of Errors

- ✓ While writing programs, very often **we get errors** in our program.
- ✓ These errors if not removed will either give **erroneous output or will not let the compiler to compile the program.**

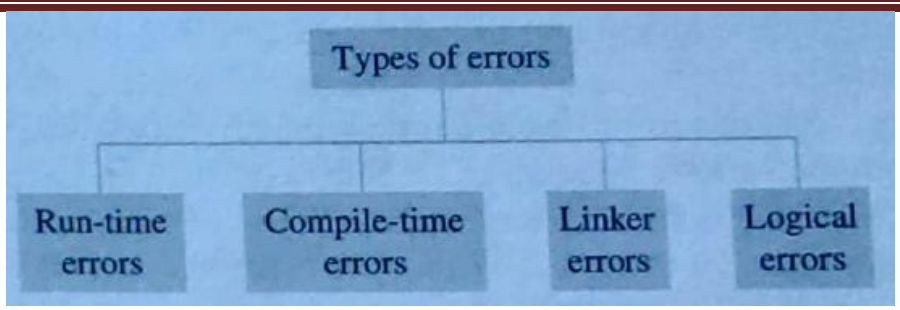


Figure 3.7. Types of Errors

1. Run-time Errors

- ✓ Run-time Errors occur when the **program is being run executed**.
- ✓ Such errors occur when the program performs **some illegal operation** like:
  - **• Dividing a number by zero.**
  - **• Opening a file that already exists.**
  - **• Lack of free memory space.**
  - **• Finding square or logarithm of negative numbers.**
- ✓ Run-time errors may **terminate program execution**, so the code must be written in such a way that it **handles all sorts of unexpected errors** rather terminating it unexpectedly.
- ✓ This **ability to continue operation of a program despite of run-time errors** is called **robustness.**

2. Compile-time Errors

- ✓ Compile-time Errors occur at the **time of compilation of the program**.
- ✓ Such errors can be further classified as follows:

(i) Syntax Errors: Syntax error is generated **when rules of C programming language** are violated. **For example**, if we write int a: then a syntax error will occur since the correct statement should be int a;

(ii) Semantic Errors: Semantic errors are those errors which **may comply with rules of the programming language but are not meaningful to the compiler.**
For **example**, if we write, a * b = c; it does not seem correct. Rather, if written like c = a * b would have been more meaningful.

3. Logical Errors

- ✓ Logical Errors are errors in the program code that **result in unexpected and undesirable output which is obviously not correct.**
- ✓ Such errors are not detected by the compiler, and **programmers must check their code line by line or use a debugger to locate and rectify the errors.**
- ✓ Logical errors occur due to incorrect statements.

For **example**, if you meant to perform c = a + b; and by mistake you typed c = a * b; then though this statement is syntactically correct it is logically wrong.

4. Linker Errors

- ✓ Linker Errors occur when **the linker is not able to find the function definition for a given prototype.**
- ✓ For example, if you write clrscr(); but do not include conio.h then a linker error will be shown.

3.5 Testing and Debugging Approaches

- ✓ Testing is an activity that is performed to verify **correct behavior of a program.**
- ✓ It is specifically carried out with an **intent to find errors.**
  - **Unit testing** is applied only on a **single unit or module to ensure whether it exhibits the expected behavior.**
  - **Integration Tests** are a **logical extension of unit tests.** In this test, two units that have already been tested are combined into a component and the interface between them is tested. This process is repeated until all the modules are tested together. The main focus of integration testing is to identify errors that occur when the units are combined.
  - **System testing checks the entire system.** For example, if our program code consists of three modules then each of the module is tested individually using unit tests and then system test is applied to test this entire system as one system.

Debugging Approaches

- ✓ Debugging is an activity that includes **execution testing and code correction.**
- ✓ It is done to **locate errors** in the program code.
- ✓ Once located, errors are then isolated and fixed to produce an **error-free code.**
- ✓ Different approaches applied for debugging a code includes:
  - **Brute-Force Method**: In this technique, a **printout of CPU registers and relevant memory locations** is **taken, studied, and documented**. It is the least efficient way of **debugging a program** and is generally done when all the other methods fail.
  - **Backtracking Method**: works by locating the **first symptom of error** and then **trace backward** across the entire source code until the real **cause of error** is detected. However, the main drawback of this approach is that with **increase in number of source code lines**, the possible backward paths become too large to manage.
  - **Cause Elimination:** lists all possible **causes of an error** is developed. Then relevant tests are carried out **to eliminate each of them**. If some tests indicate that a particular cause may be responsible for an error then the data are refined to isolate the error.

Module-1

Introduction to C

4.1 Introduction

- ✓ C is a high-level programming language.
- ✓ The programming language C was developed in the early **1970**s by **Dennis Ritchie** at **Bell Laboratories** to be used by the UNIX operating system.
- ✓ It was named C because many of its features were derived from an earlier language called B.
- ✓ Although C was designed for implementing system software, it was later on widely used for developing **portable application software**.
- ✓ C is one of the most popular programming languages.
- ✓ It is being used on several different software platforms.
- ✓ In a nutshell, there are a few computer architectures for which a C compiler does not exist.
- ✓ It is a good idea to learn C because few other programming languages such as C++ and Java are also based on C which means you will be able to learn them more easily in the future.

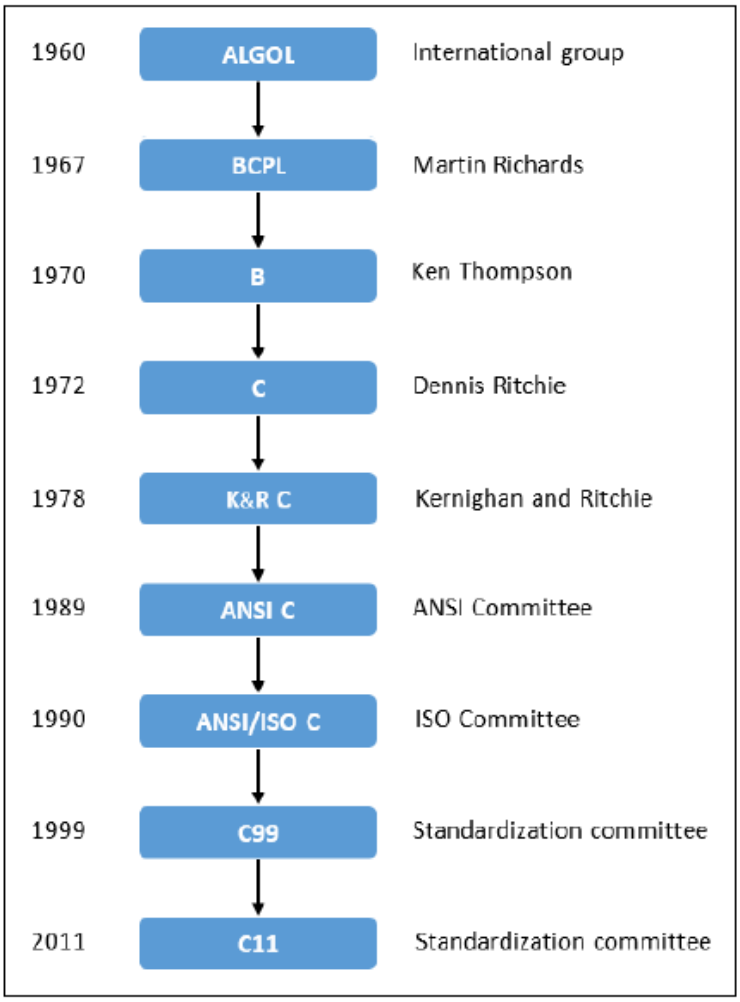


Figure 4.1. Taxonomy of C Language

4.2 Characteristics of C

C is a robust language whose rich set of **built-in functions and operators** can be used to write complex programs. The C compiler combines the features of assembly languages and high-level

languages, which makes it best suited for writing system software as well as business packages. Some basic characteristics of C language that define the language and have led to its popularity as a programming language are listed below:

- ✓ C is a **high-level programming language**, which enables the programmer to concentrate on the problem at hand and not worry about the machine code on which the program would be run.
- ✓ **Small size**: C has only 32 keywords. This makes it relatively easy to learn as compared to other languages.
- ✓ C makes extensive use of **function calls**.
- ✓ C is well suited for **structured programming**. In this programming approach, C enables users to think of a problem in terms of functions/modules where the collection of all the modules makes up a complete program. This feature facilitates ease in program debugging, testing, and maintenance.
- ✓ Unlike PASCAL it supports **loose typing** (as a character can be treated as an integer and vice versa).
- ✓ **Structured language** as the code can be organized as a collection of one or more functions
- ✓ **Stable language**: ANSI C was created in 1983 and since then it has not been revised.
- ✓ **Quick language** as a well written C program is likely to be as quick as or quicker than a program written in any other language.
- ✓ Facilitates **low-level programming.**
- ✓ **Core language:** C is a core language as many other programming languages (like C++, Java, Perl, etc) are based on C.
- ✓ C is a **portable language**, i.e., a C program written one computer can be run on another compute with little or no modification.
- ✓ C is an **extensible language** as it enables the user his own functions to the C library.
- ✓ C is often treated as the second best language for any given programming task.

4.3 Uses of C

C is a very simple language that is widely used by software professionals around the globe. The uses of C language can be summarized as follows:

- ✓ C is a very simple language that is widely used by **software professionals** around the globe.
- ✓ C is primarily used for **system programming**. The portability, efficiency, the ability to access specific hardware addresses and low runtime.
- ✓ The **compiler, libraries and interpreters** of other programming languages are often implemented in C.

- ✓ For portability and convenience reasons, C is sometimes used as an intermediate language for implementation of other languages. Major parts of popular **operating systems like windows, UNIX, Linux** are still written in C.
- ✓ C is widely used to implement **end-user applications**.
- ✓ Mobile devices like cellular phones and palmtops consisting of **microprocessor, operating system and some applications** are written in C.
- ✓ Common consumer devices like microwave ovens, washing machines and digital cameras are consisting of many **programs** which are written in C language.
- ✓ Several professional **3D computer games and many popular gaming frameworks** have been built using C language.

4.4 Basic Concepts of a C Program or Structure of a C Program

- ✓ The basic concepts of a C program can be explained by writing the structure of a C program.
- ✓ The structure of a C program is nothing but the rules that are to be followed while writing a C program.
- ✓ **The structure of a C program** is shown below:

```
[Comments]
[Preprocessor Directives]
[Global Declarations]
[Function Declarations]
[Function Definitions]
main()
{
        [Declaration Section]
        [Executable Section]
}
```

1. Comments

- ✓ **At the beginning of each program is a comment with a short description of the problem to be solved.**
- ✓ **We can use the comments anywhere in the program.**
- ✓ The comments section is **optiona**l.

  Ex: 1. /* Program1: To find the sum of two numbers*/

  2. // Program2: To calculate the area and perimeter of circle
- ✓ **The symbol /* and ends with */ represents the multiline comment.**
- ✓ **The symbol // can also be used for representing single line comment.**

2. Preprocessor directives

- ✓ **The preprocessor statements start with # symbol.**
- ✓ **These statements instruct the compiler to include some of the files in the beginning of the program.**

  Ex: #include<stdio.h>

  #include<math.h>

  are the files that the compiler includes in the beginning of the program.
- ✓ **The line containing #include<stdio.h> tells the compiler to allow our program to perform standard input and output 'stdio' operations.**

- ✓ **The '#include' directive tells the compiler that we will be using parts of the standard function library.**
- ✓ Information about these functions is contained in a series of 'header files' (stdio.h).
- ✓ .h says this file is a header file.
- ✓ The pointed brackets < and > tell the compiler the exact location of this header file.
- ✓ Using the preprocessor directives the user can define the constants also.
  Ex: #define PI 3.142

3. Global Declarations

- ✓ The variables that are declared above (before) the main program are called global variables.
- ✓ The global variables can be accessed anywhere in the main program and in all the other functions.

4. Function declarations and Definitions

- ✓ In this section the functions are declared.
- ✓ Immediately after the functions are declared, the functions can be defined.

5. The program header: main()

- ✓ **Every program must have a main function.**
- ✓ **Always the C program begins its execution from main.**

6. Body of the program

- ✓ **After the header or top lines is a set of braces ({ and }) containing a series of 'C' statements which comprise the 'body'- this is called the action portion of the program.**

Ex:
```
#include<stdio.h>
main()
{
    /* action portion of the program*/
}
```

- ✓ The body of the program contains two essential parts:
  1. Declaration section
  2. Executable section

7. Declaration section

- ✓ **The variables that are used inside the function should be declared in the declaration section.**
- ✓ For example, consider the declaration shown below:

```
int sum=0;
int a;
float b;
```

Here, the variable sum is declared as an integer variable and it is initialized to zero. The variable a is declared as an integer variable whereas the variable b is declared as a floating point variable.

8. Executable section

- ✓ **They represent the instructions given to the computer to perform a specific task.**

- ✓ **The instructions can be input/output statements, expressions to be evaluated, simple assignment statements, control statements such as if statement, for statement etc.**
- ✓ **Each executable statement ends with ";".**

Example: Write a C program to display "Hello World".

```
#include<stdio.h>
void main()
{
    printf("Hello World");
}
```

4.5 Files used in a C Program

A C program uses four types of files as follows:

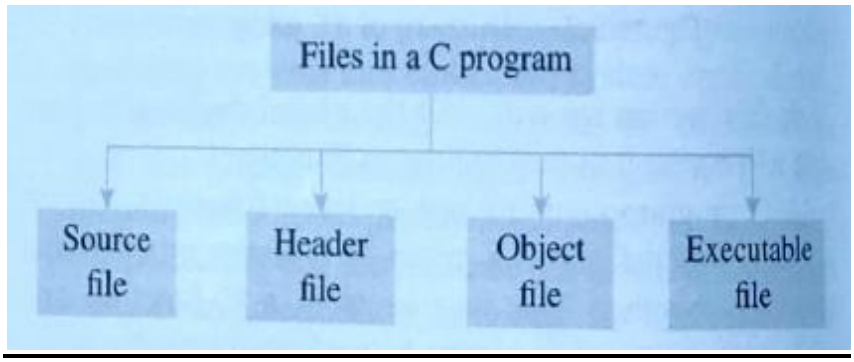


Figure 4.2. Files in a C program

1. Source Code File

- ✓ This file includes the **source code of the program**.
- ✓ The **extension** for these kind of files are **'.c'**.
- ✓ It defines the **main and many more functions** written in C.
- ✓ **main() is the starting point of the program**. It may also contain **other source code files**.

2. Header Files

- ✓ They have an **extension '.h'.**
- ✓ They contain the **C function declarations and macro definitions** that are shared between various source files.

Advantages of header files:

1. At times the programmer may want to use the same **subroutines for different programs**. To do this, he would just **compile the code of the subroutine once and link to the resulting object file in any file** in which the functionalities of this subroutine are required.

2. At times the **programmer may want to change or add the subroutines and reflect those changes in all the programs.** For doing this, he will have to only change the source file for the subroutines, recompile the source code and then recompile and re-link the program. This tells us that including a header file will make it easier at all levels of the program. If we need to modify anything then changes are made only in the subroutines after which all the changes will be reflected.

Standard header files

- ✓ C provides us with some standard header files which are available easily.

Common standard header files are:

- ✓ **string.h –** used for handling string functions.
- ✓ **string.h –** used for handling string functions.
- ✓ **stdlib.h –** used for some miscellaneous functions.
- ✓ **stdio.h –** used for giving standardized input and output.
- ✓ **math.h –** used for mathematical functions.
- ✓ **alloc.h –** used for dynamic memory allocation.
- ✓ **conio.h –** used for clearing the screen.

- ✓ The header files are added at the **start of the source code** so that they can be used by more than one function of the same file.

3. Object files

- ✓ They are the files that are generated by the **compiler as the source code file is processed**.
- ✓ These files generally contain the **binary code of the function definitions**.
- ✓ The object file is used by the **linker for producing an executable file** for combining the object files together. It has a **'.o' extension**.

4. Executable file

- ✓ This file is generated by the **linker.**
- ✓ **Various object files are linked by the linker** for producing a **binary file** which will be executed directly.
- ✓ They have an **'.exe' extension**.

4.6 Compiling and Executing C Programs

- ✓ C is a **compiled language.**
- ✓ The programming process starts with creating a **source file** that consists of the statements of the program written in C language. This source file usually contains ASCII characters and can be produced with a text editor, such as Windows notepad, or in an Integrated Design Environment.
- ✓ The source file is then processed by a special program called a **compiler.**
- ✓ **The compiler translates the source code into an object code.**
- ✓ The object code contains the **machine instructions** for the CPU, and calls to the operating system API (Application Programming Interface).
- ✓ However, even the object file is not an executable file. Therefore, in the next step, **the object file is processed with another special program called a linker**.
- ✓ While there is a different compiler for every individual language, the same linker is used for object files regardless of the original language in which the new program was written. The output of the linker is an executable or runnable file. The process is shown in Figure 4.3.

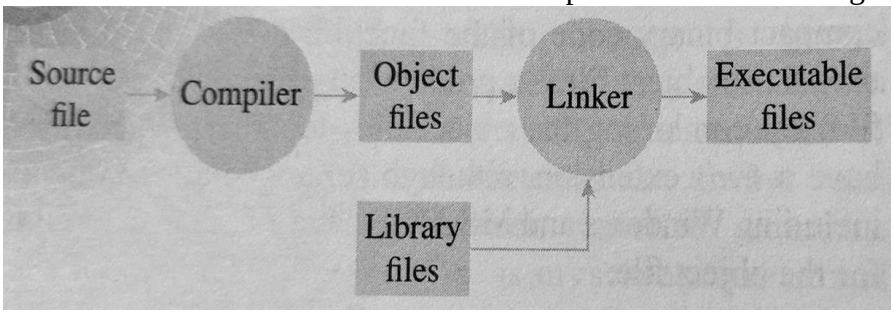


Figure 4.3. Overview of compilation and execution process

- ✓ In C language programs, there are two kinds of source files. In addition to the **main (.c) source file**, which contains executable statements there are also **header (.h) source files**.
- ✓ Every C program uses standard header files, which are written as part of the source code for **modular C programs**.
- ✓ The compilation process shown in Figure 4.4 is done in two steps. In the first step, the **preprocessor program** reads the source file as text, and produces another text file as output. The output of the preprocessor is a text file which does not contain any preprocessor statements. This file is ready to be processed by the compiler.

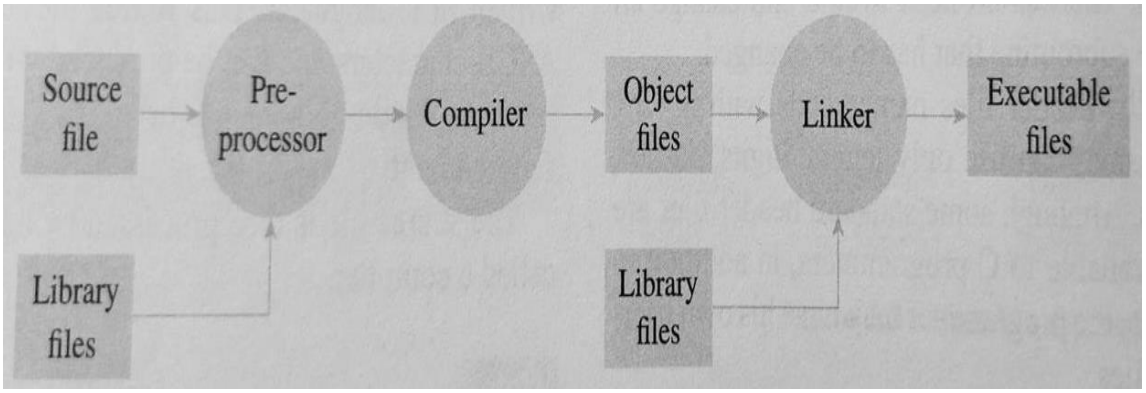


Figure 4.4. Preprocessing before compilation

- ✓ **The linker combines the object file with library routines (supplied with the compiler) to produce the final executable file.**
- ✓ In modular programming, the source code is divided into two or more source files. All these source files are compiled separately thereby producing multiple object files. These object files are combined by the linker to produce an executable file (Figure 4.5).

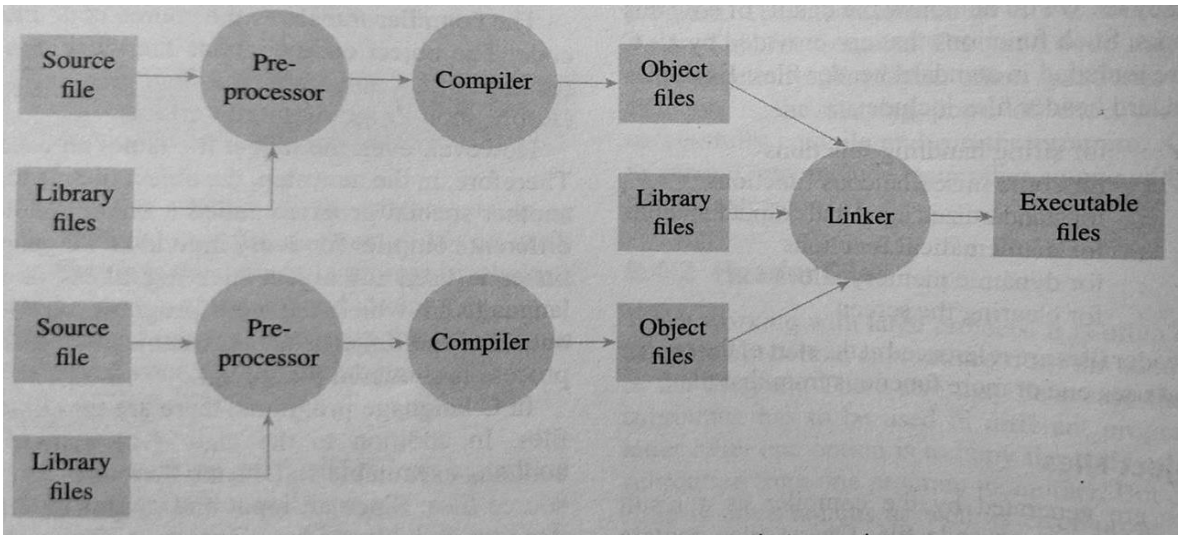


Figure 4.5. Modular programming-the complete compilation and execution process

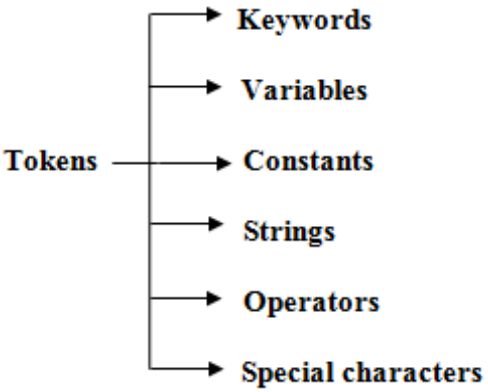
4.7 Using Comments

- ✓ Many a time the meaning or the purpose of the file code is not clear to the reader.
- ✓ Therefore it is a **good programming practice to place some comments in the code to help the reader understand the code clearly**.
- ✓ Comments are just a **way of explaining what a program does**.
- ✓ It is merely an internal program documentation.

- ✓ The **compiler ignores the comments when forming the object file means that the comments are non-executable statements.**
- ✓ **C supports two types of comments.**
  // is used to comment a **single statement**. This is known as a **line comment**. A line comment can be placed anywhere on the line and it does not require to be specifically ended as the end of the line automatically ends the line.
  /* is used to comment **multiple statements**. A **/*** is ended with ***/** and all statements that lie within these characters are commented. This type of comment is known as block comment.

4.8 C Tokens

- ✓ **Tokens are the basic building blocks in C language.**
- ✓ **A token is the smallest individual unit in a C program.**
- ✓ This means that a program is constructed using a combination of these tokens.
- ✓ There are six main types of tokens in C as shown below:



1. Keywords:

- ✓ **Keywords are the tokens which have predefined meaning in C language, whose meaning cannot be changed by the user.**
- ✓ All keywords are basically a **sequence of characters** that have a **fixed meaning.**
- ✓ They are also called **reserved** words.
- ✓ C Keywords are case sensitive. Therefore all C keywords must be written in small letters.

  int, float, if, while, void etc are valid keywords.

  Int, Float, IF, VOID etc are invalid keywords. (They are written using capital letters).

List of keywords:

- ✓ There are total 32 keywords in C language as shown below.

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

2. Variables: A variable is a **data item whose value changes during the execution of program.**
Ex: int a,b,sum;

3. Constants: A constant is a **data item which will not change during the execution of a program.**
Ex: #define PI 3.142

4. Strings: String is an **array of characters and terminated by NULL character** which is denoted by **'\0'.**
Ex: char name[21];

5. Operators: Operator is a **symbol (or token) that specifies the operation to be performed on various types of data.**
Ex: Arithmetic operators (+,-,*,/), Relational operators (&&,||,!), Logical operators (>,>=,<,<=) and Assignment operator (=) etc.

6. Special characters: [ ], { }, ( ) etc. used in the program to execute the code correctly and helps to write a complex codes by special symbols.

4.9 CHARACTER SET IN C

- ✓ Like in natural languages, computer languages also use a **character set that defines the fundamental units used to represent information.**
- ✓ In C, a character means any letter from **English alphabet, a digit or a special symbol** used to represent information.
- ✓ These characters when combined together form tokens that act as basic building blocks of a C program.
- ✓ The character set of C can therefore be given as:
  a. English alphabet: Include both lower case (a z) as well as upper case (A Z) letters
  b. Digits: Include numerical digits from 0 to 9
  c. Special characters: Include symbols such as, % &) < > * S /) [" etc.,
  d. White space characters: These characters are used to print a blank space on the screen. They are shown in Figure 4.6.

White space character	Meaning
\b	Blank space
\t	Horizontal tab
\v	Vertical return
\r	Carriage return
\f	Form feed
\n	New line

Figure 4.6. White space characters in C

e. Escape sequence: '\n' is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Like the newline character, the other escape sequences supported by C language are shown in Table 4.1.

Table 4.1: Escape sequences

Escape sequence	Purpose	Escape sequence	Purpose
\a	Audible signal	\?	Question mark
\b	Backspace	\\	Back slash
\t	Tab	\'	Single quote
\n	Newline	\"	Double quote
\v	Vertical tab	\0	Octal constant
\f	New page\ Clear screen	\x	Hexadecimal constant
\r	Carriage return		

4.10 Identifiers

- ✓ Identifiers help us to **identify data and other objects in the program.**
- ✓ Identifiers are basically the **names given to program elements such as variables, arrays, and functions.**
- ✓ Identifiers may consist of sequence of **letters, numerals, or underscores**.

Rules for Forming Identifier Names

Some rules have to be followed while forming identifier names. They are as follows:

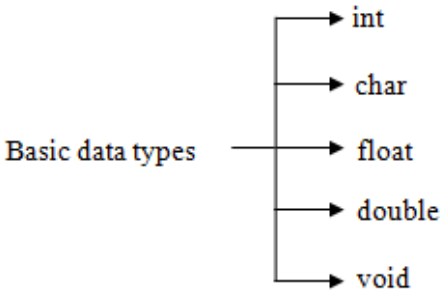
- ✓ Identifiers cannot include any **special characters or punctuation marks (like #, s, ^, ?, ., etc.) except the underscore '_'.**
- ✓ There **cannot be two successive underscores.**
- ✓ **Keywords cannot be used** as identifiers.
- ✓ The **case of alphabetic characters** that form the identifier name is **significant**. For example, 'FIRST' is different from 'first' and 'First'.
- ✓ Identifiers must **begin with a letter or an underscore.** However, use of underscore as the first character must be avoided because several compiler-defined identifiers in the standard C library have underscore as their first character.
- ✓ Identifiers can be of any reasonable length. They should **not contain more than 31 characters.** They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.

Examples of valid identifiers include:
roll_number, marks, name, emp_number, basic_pay, HRA, DA, dept_code, DeptCode, RollNo, EMP_NO

Examples of invalid identifiers include:
23 student, %marks, @name, #emp_number, basic.pay, -HRA, (DA), &dept_code, auto

4.11 Basic Data types in C and Sizes

- ✓ **The data type defines the type of data stored in a memory location.**
- ✓ **The data type determines how much memory should be allocated for a variable.**
- ✓ The data types that can be manipulated by machine instructions are called 'basic or primitive data types'.



i. int:

- ✓ **An int is a keyword which is used for defining integers in C language**.
- ✓ **Using int the programmer can inform the compiler that the data associated with this should be treated as integer**.
- ✓ **Using 'int' compiler determines the size of the data (2 bytes) and reserve space in memory to store the data.**
- ✓ **Integer data types namely:**

1. short int **2. int** **3.long int**

Type	Size
short int	2 bytes
int	2 bytes
long int	4 bytes

Ex: int a,b,c;

ii. float:

- ✓ **A float is a keyword which is used to define floating point numbers in C language.**
- ✓ **The programmer can inform the compiler that the data associated with this keyword should be treated as floating point number**.
- ✓ The default precision of floating point number is 6 digits after dot(.).

	Size of float
16-bit Machine	4 bytes
32-bit Machine	8 bytes

Ex: float x,y,z;

iii. double:

- ✓ **It is a keyword which is used to define long floating point numbers in C language.**
- ✓ The default precision of floating point number is 14 digits after dot(.).

	Size of double
16-bit Machine	8 bytes
32-bit Machine	16 bytes

Ex: double p,q,r;

iv. char:

- ✓ **It is a keyword which is used to define single character or a sequence of characters called String in C language.**
- ✓ **Using this keyword, the compiler determines the size of the data and reserve space in memory to store the data.**
- ✓ Each character stored in the memory is associated with a unique value called an ASCII (American Standards Code for Information Interchange).

	Size of char	Range of Unsigned char	Range of Signed char
16/32-bit Machine	1 byte	0 to 255	-128 to +127

Ex: char ch; // ch variable stores a single character Ex: ch= 'a';
char s[20]; // s variable stores a string(group of characters) Ex: s= "jitdvg";

v. void:

- ✓ **It is an empty data type, since no value is associated with this data type.**
- ✓ **It does not occupy any space in the memory.**

	Size of void	Range
16/32-bit Machine	0	No value

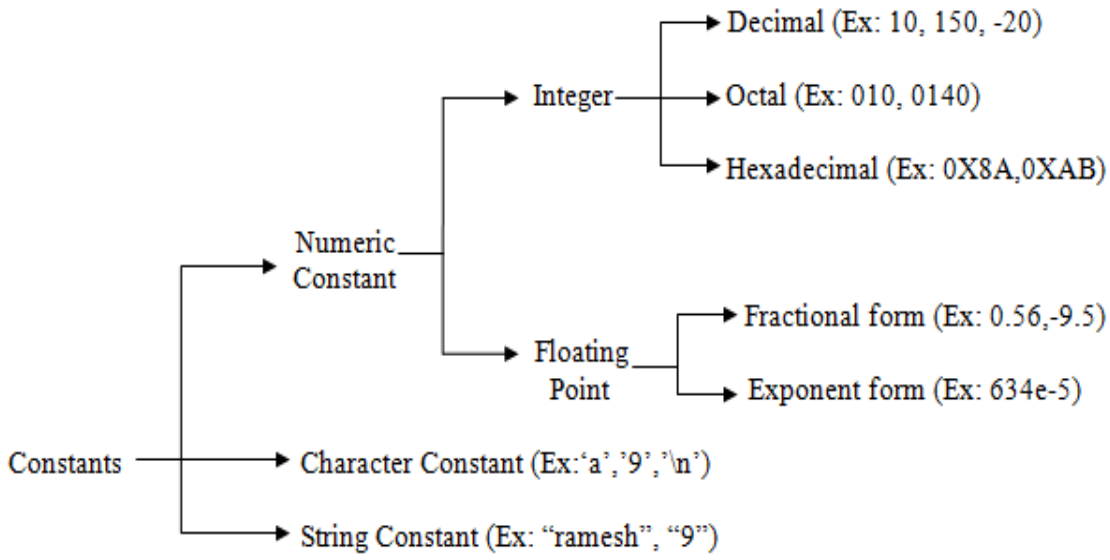
Ex:
```
void main( )
{

}
```

It is primarily used in three cases:

- ✓ To specify the **return type of a function** (when the function returns no value)
- ✓ To specify the **parameters of the function** (when the function accepts no arguments from the caller)
- ✓ To create **generic pointers**.

4.12 Constants

- ✓ **A constant is a data item which will not change during the execution of a program.**
  or
  Constants are identifiers whose values do not change.
- ✓ While values of variables can be changed at any time, values of constants can never be changed. Constants are used to define fixed values like mathematical constant pi or the charge on an electron so that their value does not get changed in the program even by mistake.
- ✓ C allows the programmer to specify constants of integer type, floating point type, character type, and string type.
- ✓ The **constants cannot be modified** in the program.



4.12.1 Types of Constants

1. Integer Constants

- ✓ A constant of integer type consists of a sequence of digits.
- ✓ For example, 1, 34, 567, 8907 are valid integer constants.
- ✓ Integer literals can be expressed in **decimal, octal or hexadecimal notation**.
- ✓ By default an integer is expressed in decimal notation. Decimal integers consists of a set of **digits 0 through 9, preceded with an optional + or – sign**. Example: 123, -123, +123, and 0.
- ✓ An integer constant **preceded by a zero (0) is an octal number**. Octal integers consist of a set of digits, **0 through 7**. Example: 012, 0, 01234.
- ✓ An integer constant is expressed in **hexadecimal notation if it is preceded with 0x or 0X.** Hexadecimal numbers contain **digits from 0-9 and letters A through F**, which represent numbers 10 through 15. Example: Ox12, Ox7F, OxABCD, Ox1A3B.

2. Floating Point Constants

- ✓ A floating point constant consists of an **integer part, a decimal point, a fractional part, and an exponent field containing an e or E** (e means exponent) followed by an integer where the fraction part and integer part are a sequence of digits. However, it is not necessary that every floating point constant must contain all these parts. Some floating point numbers may have certain parts missing. Some valid examples of floating point numbers are: 0.02, -0.23, 123.456, +0.34 123, 0.9, -0.7, +0.8 etc.
- ✓ To make it a **float type literal**, you must specify it using **Suffix F or f**. Consider some valid floating point literals given below. (Note that suffix L is for 1ong double.)
  0.02F 0.34f 3.141592654L 0.002146 2.146E-3
- ✓ A floating point number may also be expressed in **scientific notation**. Therefore, the numbers given below are valid floating point numbers:
  0.5e2 14E-2 1.2e+3 2.1E-3 -5.6e-2

3. Character Constants

- ✓ A character constant consists of a **single character enclosed in single quotes**.
- ✓ For example, 'a' and '@' are character constants.
- ✓ In computers, characters are stored using machines character set using **ASCII codes**. All escape sequences mentioned in Table 4.1 are also character constants.

4. String Constants

- ✓ A string constant is in **double quotes**.
- ✓ So "a" is not the same as 'a'.
- ✓ The characters comprising the string constant are stored in **successive memory locations**.
- ✓ When a string constant is encountered in a C program, the compiler records the address of the first character and appends a null character ('\0') to the string to mark the end of the string. Thus, length of a string constant is equal to number of characters in the string plus 1 (for the null character). Therefore, the length of string literal "hello" is 6.

4.12.2 Declaring Constants

- ✓ To declare a constant, precede the normal variable declaration with "**const" keyword** and assign it a value. For example,
  const float pi = 3.14;
  The const keyword specifies that the value of pi cannot change.
- ✓ However, another way to designate a constant is to use the pre-processor command **define**.
  Example: #define PI 3.14159
  #define service_tax 0.12
  In these examples, the value of pi will never change but service tax may change. Whenever the value of the service tax is altered, it needs to be corrected only in the define statement.
  Some rules that need to be applied to a #define statement which defines a constant:
  Rule 1: Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters.
  Rule 2: No blank spaces are permitted between the # symbol and define keyword.
  Rule 3: Blank space must be used between #define and constant name and between constant name and constant value.
  Rule 4: #define is a pre-processor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

4.13 Variable

- ✓ **A variable is a name given to a memory location within the computer that can hold one value at a time.**

  OR

  A variable is a data item whose value changes during the execution of program.
- ✓ Every variable should be associated with **type, size and value.**
- ✓ Whenever a new value is placed into a variable, it replaces the previous value.

4.13.1 Rules for defining variables

Some rules have to be followed while forming variable names. They are as follows:

- ✓ Variables cannot include any **special characters or punctuation marks (like #, s, ^, ?, ., etc.) except the underscore '_'.**
- ✓ There **cannot be two successive underscores.**
- ✓ **Keywords cannot be used** as variables.

- ✓ The **case of alphabetic characters** that form the variable name is **significant**. For example, 'FIRST' is different from 'first' and 'First'.
- ✓ Variables must **begin with a letter or an underscore.**
- ✓ Variables can be of any reasonable length. They should **not contain more than 31 characters.** They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.

  Ex: Classify the following into valid and invalid variable names in C. If invalid give reasons:

Variable	Valid/Invalid	Reason
principle_amount	Valid	
A	Valid	
Sum1	Valid	
for1	Valid	
if	Invalid	It is a keyword
for	Invalid	It is a keyword
3_factorial	Invalid	Should not start with digit
Sum,1	Invalid	Comma should not be there
sum-of-digits	Invalid	Minus sign should not be there
sum_of_digits	Valid	
sum of digits	Invalid	No spaces are allowed
$sum	Invalid	$ sign should not be there
sum=	Invalid	= sign should not be there
one+two	Invalid	+ sign should not be there
sum!	Invalid	! sign should not be there
int	Invalid	It is a keyword
$roll no	Invalid	$ sign should not be there
_name1	Valid	
James bond	Invalid	No spaces are allowed

4.13.2 Types of Variables

C language supports two basic kinds of variables numeric and character.

1. Numeric Variables

- ✓ Numeric variables can be used to **store either integer values or floating point values**.
- ✓ While an **integer value is a whole number without a fraction part or decimal point, a floating point value can have a decimal point.**
- ✓ Numeric variables may also be associated with modifiers, such as **short, long, signed, and unsigned**. The difference between signed and unsigned numeric variables is that **signed variables can be either negative or positive** but **unsigned variables can only be positive**.
- ✓ When we do not specify the signed/ unsigned modifier, **C language automatically takes it as a signed variable.** To declare an unsigned variable, the unsigned modifier must be explicitly added during the declaration of the variable.

2. Character Variables

- ✓ Character variables are just **single characters enclosed within single quotes**. These characters could be any character from the ASCII character set-letters ('a', 'A'), numerals ('2'), or special characters ('&').
- ✓ A number that is given in single quotes is not the same as a number without them. This is because 2 is treated as an integer value, but '2' is considered character not an integer.

4.13.3 Declaring/Defining a Variable

- ✓ Each variable to be used in the program must be declared.
- ✓ 'Declaration' tells the computer which storage locations or variables to use in the program.
- ✓ **It is a method of informing the compiler to reserve the memory space for the program data based on the type of variables.**
- ✓ To declare a variable, specify the data type of the variable followed by its name.
- ✓ The **data type** indicates the **kind of values that the variable will store**.

Syntax:

datatype v_1,v_2,…….,v_n;

Where, datatype: it can be int, float, char, double etc.

v_1,v_2,---,v_n: is a list of variables, which are separated by commas.

Ex: int a,b,c;
float p,t,r,si;

4.13.4 Initializing the variables

- ✓ **Initialization is the process of assigning values to the variables.**

Syntax:

datatype var_name=data;

Where, datatype: is the type of the data to be stored in memory location. (int,float,char,double)
var_name: name of a variable
= is assignment operator
data: is the value to be stored in memory associated with variable var_name.

Ex: int a=10;

4.14 Input/Output statements in C

- ✓ A Stream is the **source of data** as well as the **destination of data**.
- ✓ Streams are associated with a **physical device** such as a **monitor or a file** stored on the secondary memory.
- ✓ C uses two forms of streams:
  - **Text stream**: In a text stream, sequence of characters is divided into lines, each line being terminated with a new character(\n).
  - **Binary stream**: A binary stream contains data values using their memory representation.

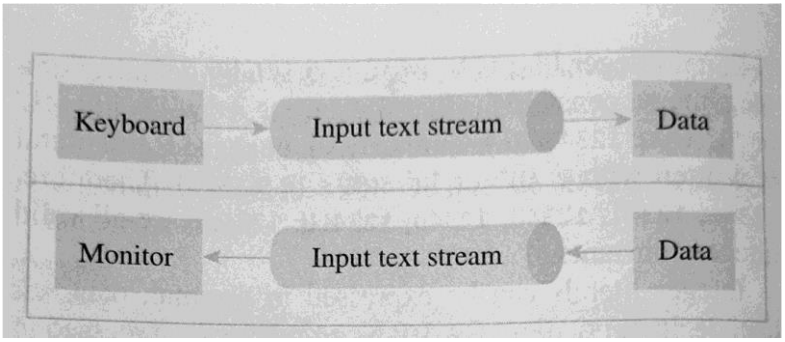


Figure 4.7. Input and output streams in C

Formatting Input/Output

- ✓ C language supports two formatting functions **printf and scanf**.
- ✓ **printf is used to convert data stored in the program into a text stream for output to the monitor and scanf is used to convert the text stream coming from the keyboard to data values and stores them in program variables.**

1. printf()

- ✓ The printf() function (stands for print formatting), is used to **display information required by the user and also prints the values of variables.**
- ✓ For this, **the printf() function takes data values, converts them to a text stream using formatting specifications in the control string and passes the resulting text stream to standard output.**
- ✓ Each data value to be formatted in the text stream is described using a separate conversion specification in the control string.
- ✓ The **specification in the control string** describes **data value's type, size, specific format.**

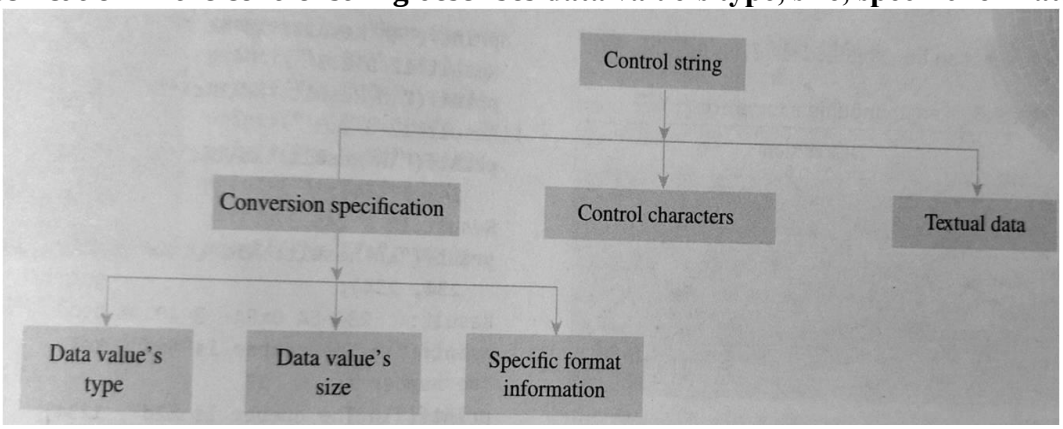


Figure 4.8. printf() in C

The syntax of printf function:

printf("control string", variable list);

- ✓ The function accepts two parameters- **control string & variable list**.
- ✓ The control string may also contain the text to be printed like instructions to the user, captions, identifiers or any other text to make the output readable.
- ✓ In some printf statements there may be only a text string that has to be displayed on the screen.
- ✓ The control characters can also be included in the printf statement like \n, \t, \a, etc.
- ✓ The **prototype of control string**:

%[flags][width[[.precision][length modifier] type specifier

- ✓ Each control string must begin with a % sign. The % sign specifies how the next variable in the list of variables has to be printed.
- ✓ After % sign follows:

Flags

- ✓ Flag is an optional argument which specifies output justifications such as numerical sign, trailing zeros or octal, decimal or hexadecimal prefixes.

Types of flags:

-	:	Left justify
+	:	Display data with numeric sign
#	:	Provide additional specifier o,O,X,0,0x
0	:	Left-padding with zeros

Width

- ✓ It is an optional argument which specifies minimum number of positions in the output.

Precision

- ✓ Precision is an optional argument which specifies the maximum number of characters to print.

- ✓ Therefore, a conversion specification %7.3f means print a floating point value of maximum 7 digits where 3 digits are allotted for the digits after the decimal point.

Length modifiers

Table 4.2 Length modifiers for printf()

Length	Description
h	When the argument is a short int or unsigned short int
l	When the argument is a long int or unsigned long int (used for integer specifiers)
L	When the argument is a long double (used for floating point specifiers)

Type specifiers

- ✓ Type specifiers are used to define the type and interpretation of the value of the corresponding argument.

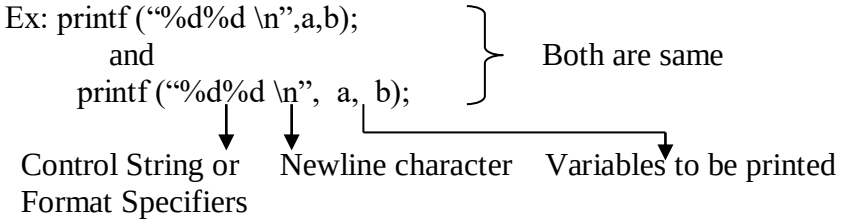
Table 4.3 Type specifiers for printf()

Type	Qualifying input
c	For single characters
d	For integer values
f	For floating point numbers
E, e	Floating point numbers in exponential format
G, g	Floating point numbers in the shorter of e format
o	For octal numbers
s	For a sequence of (string of) characters
u	For unsigned integer values
X, x	For hexadecimal values

Guidelines/Rules for printf()

- ✓ A printf() always contains a string or format string in quotation marks.
- ✓ The control string may or may not be followed by some variables or expressions whose value we want printed.
- ✓ Each value to be printed needs a 'conversion specification' like %d to hold its place in the control string.
- ✓ This conversion specification describes the exact way the value is to be printed.
- ✓ When printf() is executed each conversion specification is replaced by the value of the corresponding expression, then print according to the rules in specification.
- ✓ The symbols \n or \t in control string tell the machine to skip to new line or tab. It affects the appearance of the output but not displayed as part of it.
- ✓ A word or blank space or punctuation symbols within the control string will print exactly as it appears.

- ✓ If there are variables or expressions to be printed, commas are used to separate them from the control string and each other, once comma is used as separator, it is not necessary to add blank spaces (not allowed).



2. scanf()

- ✓ The scanf() function stands for **scan formatting** and **is used to read formatted data from the keyboard.**
- ✓ **The scanf() function takes a text stream from the keyboard, extracts and formats the data from the stream according to a format control string and then stores the data in a specified program variables.**
- ✓ Syntax of the scanf() function:

  scanf ("control string", arg1, arg2, arg3,….. argn);
- ✓ Control string specifies the type and format of the data that has to be obtained from the keyboard and stored in the memory locations pointed by arg1, arg2, …, argn,i.e., arguments are actually the variable addresses where each piece of data is to be stored.
- ✓ Prototype of the control string can be given as:

  %[*][width][modifier] type
- ✓ Here the * is an optional argument which indicates that data should be read from the stream, but ignored (not stored in memory location).

Width

- ✓ This is an optional argument that specifies the maximum number of characters to be read.
- ✓ Fewer characters are read if scanf() encounters a white space and will stop processing further.

Modifier

- ✓ It is an optional argument that can be h, l, or L for the data pointed by corresponding additional arguments.
- ✓ Modifier h is used for short int or unsigned short int, l is used for long int, unsigned long int, or double values. Finally, L is used for long double data values.

Type

- ✓ It specifies the type of data that has to be read.
- ✓ The type specifiers for scanf( ) function are same as that of printf ( ) function.
- ✓ The scanf function ignores any blank spaces, tabs and newlines entered by the user.
- ✓ The address of the variable is denoted by an '&' sign followed by the name of the variable.

Rules to use a scanf function:

Rule 1: The scanf function works until:

- the maximum number of characters has been processed
- a white space character is encountered,
- or an error is detected.

Rule 2: Every variable that has to be processed must have a conversion specification associated with it. Therefore, following scanf statement will generate an error as num3 has no conversion specification associated with it.

```
scanf(" %d %d", &num1, &num2, &num3);
```

Rule 3: There must be a variable address for each conversion specification. Therefore following scanf will generate an error as no variable address is given for the third conversion specification.

```
scanf(%d %d %d", &num1, &num2);
```

Rule 4: An error will be generated if the format string is ended with the white space character.

Rule 5: The data entered by the user must match the character specified in the control string, otherwise, an error will be generated and scanf will stop its processing.
For example, consider the following scanf of statement

```
scanf("%d / %d", &num1, &num2);
```

Here the slash in the constant string is neither a white space character nor a part of the conversion specification, so the users must enter data of the form 21/46.

Rule 6: Input data values must be separated by spaces.

Rule 7: Any unread data value will be considered as a part of data input in the next call to the scanf.

Rule 8: When the field width specifier is used, it should be large enough to contain the input data size.

Examples of printf/scanf:

1. Code to input values in variables of different data types:

- int num;
  scanf("%d",&num);

- float salary;
  scanf("%f", &salary);

- char ch;
  scanf("%c",&ch);

- char str[10];
  scanf("%s", str);

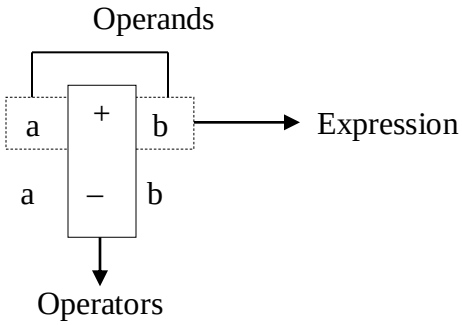
2. Reading variables of different data types in one statement:

```
int num;
float fnum;
char ch;
char str[10];
scanf("%d %f %c %s", &num, &fnum, &ch, str);
```

MODULE-2

2.1 Operators and Expressions

- ✓ **Operator: Operator is a symbol (or token) that specifies the operation to be performed on various types of data.**
- ✓ **Operand: A constant or variable or function which returns a value is an operand.**
- ✓ **Expression: A sequence of operands and operators that reduces to a single value is an expression.**

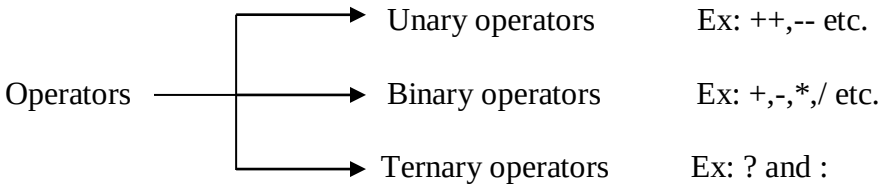


2.1.1 Classification of operators

The operators in C can be classified based on:

- The number of operands an operator has.
- The type of operation being performed.

1. Classification of operators based on the number of operands

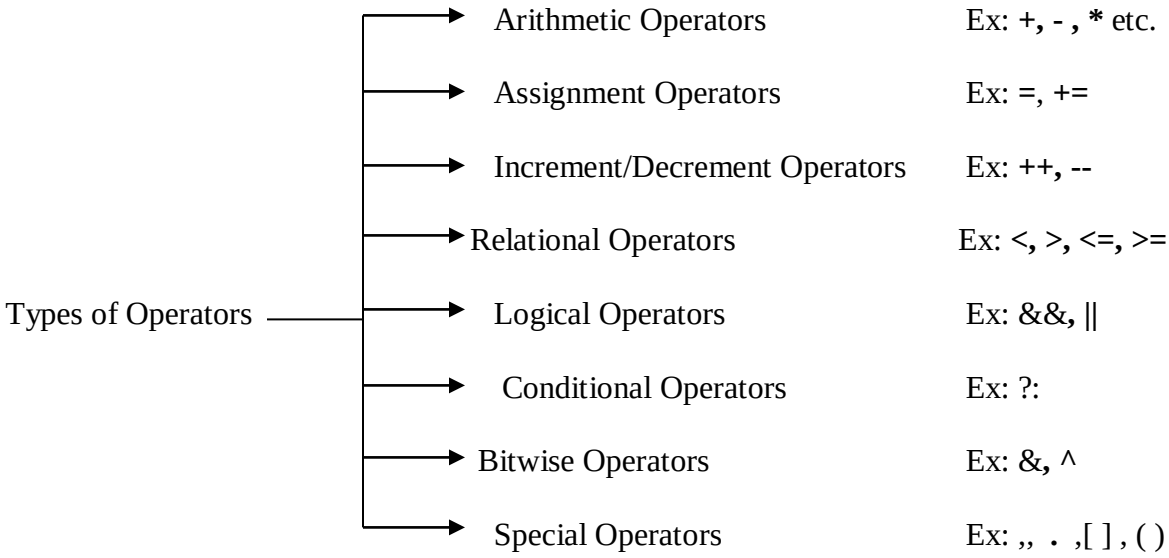


i) **Unary operator:** An operator which acts on only one operand to produce the result is called Unary operator.
Ex: -10, -a, *b, ++a, a++, b-- etc.

ii) **Binary operator:** An operator which acts on two operands to produce the result is called Binary operator.
Ex: a+b, a*b, 10/5 etc

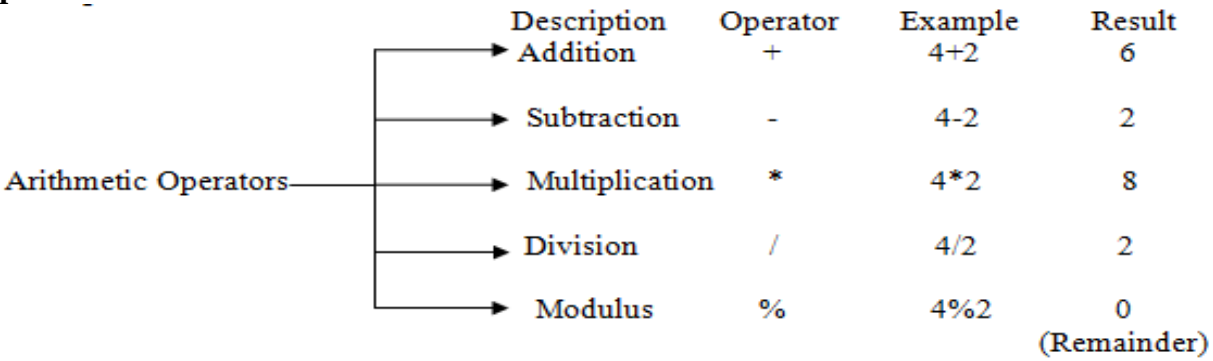
iii) **Ternary operator:** An operator which acts on three operands to produce the result is called Ternary operator.
Ex: a ? b : c;

2. Classification of operators based on type of operation

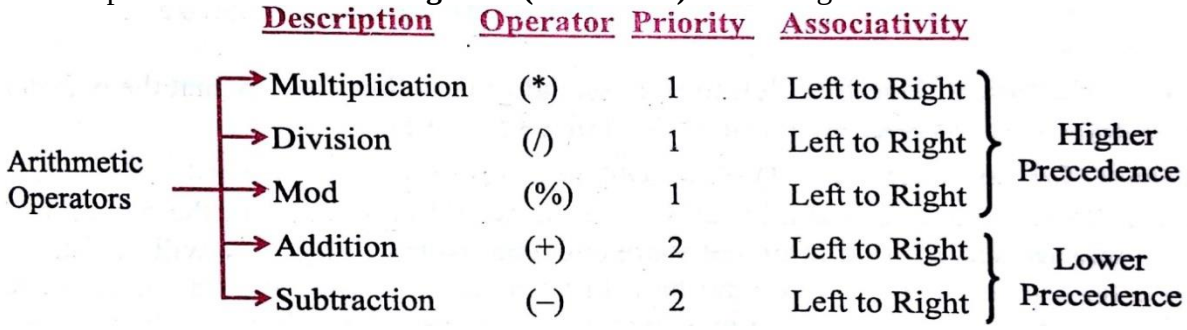


i) Arithmetic Operators

- ✓ **The operators that are used to perform arithmetic operations such as addition, subtraction, multiplication, division and modulus are called arithmetic operators.**
- ✓ These operators perform operations on two operands and hence they are called **binary operators.**



- ✓ **% (modulus operator)** divides the first operand by second and returns the **remainder.**
- ✓ % (modulus operator) cannot be applied to floating or double.
- ✓ % operator returns **remaining value(remainder)** of an integer division.



- ✓ *, / and % operators are having higher precedence than +, - operators.

Arithmetic Operator's Precedence (Precedence of operators)

- ✓ The Arithmetic Expressions are evaluated based on **"BODMAS" Rule**. (Brackets of Operator (x(2),[ ]), Order or Power (x^y,2^3 etc.), Division, Multiplication, Addition, Subtraction).

Now, let us see what is the result of 6 x (2 + 3) / 5. Based on the BODMAS rule (priority), we evaluate the given expression as shown below:

Step 1:	6 x (2 + 3) / 5	Brackets have the first precedence
Step 2:	6 x 5 / 5	Division has the second precedence
Step 3:	6 x 1 → 6	Multiplication has the third precedence

Example 9: Now, let us see what is the result of **8 + 4* 3**

The given expression is evaluated as shown below:

8 + 4 * 3 Multiplication has highest precedence

8 + 12 Addition has least precedence

20

ii) Assignment Operators

- ✓ **An operator which is used to assign the data or result of an expression into a variable (also called memory location) is called an assignment operator.**
- ✓ Assignment operator is denoted by '=' sign.

Ex: a=b; //value of b is copied into variable a

Types of Assignment statements:

a). Simple Assignment Statement

```
variable= expression;
```

Ex: a=10;
a=b;
a=a+b;
Area= l*b; //Result of Expression l*b is copied into variable area.

b). Shorthand Assignment Statement

- ✓ **The operators such as +=, -=, *=, /= and %= are called shorthand assignment operators.**

Ex 1: a=a+10; // simple assignment
a+=10; // shorthand assignment
Ex 2: i=i+2;
i+=2;

- ✓ If $expr_1$ and $expr_2$ are expressions then

$expr_1 op = expr_2$;

is equivalent to

$expr_1 = (expr_1)\ op\ (expr_2)$;

i.e., x*=y+1 means x=x*(y+1) ;

Simplify the expression: x *= y + 3
when $x = 10$ and $y = 5$

Solution: The given expression is:

x *= y + 3

can be written as

x *= (y + 3)

which in turn can be interpreted as:

x = x * (y + 3)

x = 10 * (5 + 3) (after substituting for x and y)

x = 10 * 8 = 80

So, **Result = 80**

c). Multiple Assignment Statement

- ✓ Assigning a value or a set of values to different variables in one statement is called multiple assignment statement.
- ✓ The multiple assignments are used whenever same value has to be copied into various memory locations.

Ex: int i=10; //simple assignment
int j=10;
int k=10;
int i=j=k=10; //multiple assignment

iii) Increment and Decrement Operators

Increment Operator

- ✓ **'++' is an increment operator. This is unary operator. It increments the value of a variable by one.**

Increment Operator → Post Increment Ex: a++
Increment Operator → Pre Increment Ex: ++a

1. Post Increment

- ✓ **It increments the value after (post) the operand value is used. i.e., operand value is used first and then the operand value is incremented by 1.**

```
Ex: void main()
    {
        int a=20,b;          // a=20, b?
        b=a++;               // b=a=20, a=a+1= 20+1
        printf("%d",a);      // a=21
        printf("%d",b);      // b=20
    }
```

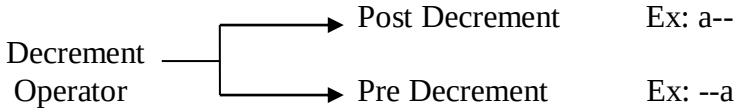
2. Pre Increment

- ✓ **It increments before (pre) the operand value is used. The operand value is incremented by 1 and this incremented value is used.**

```
Ex:  void main()
     {
          int a=20,b;          // a=20, b?
          b=++a;               // a=++a=a+1= 20+1, b=a=21
          printf("%d",a);      // a=21
          printf("%d",b);      // b=21
      }
```

Decrement Operator

- ✓ **'--' is a decrement operator. This is a unary operator. It decrements the value of a variable by one.**



1. Post Decrement

- ✓ **It decrements the value after (post) the operand value is used. i.e., operand value is used first and then the operand value is decremented by 1.**

```
Ex: void main()
    {
      int a=20,b;          // a=20, b?
      b=a--;               // b=a=20, a=a-1= 20-1
      printf("%d",a);      // a=19
      printf("%d",b);      // b=20
     }
```

2. Pre Decrement

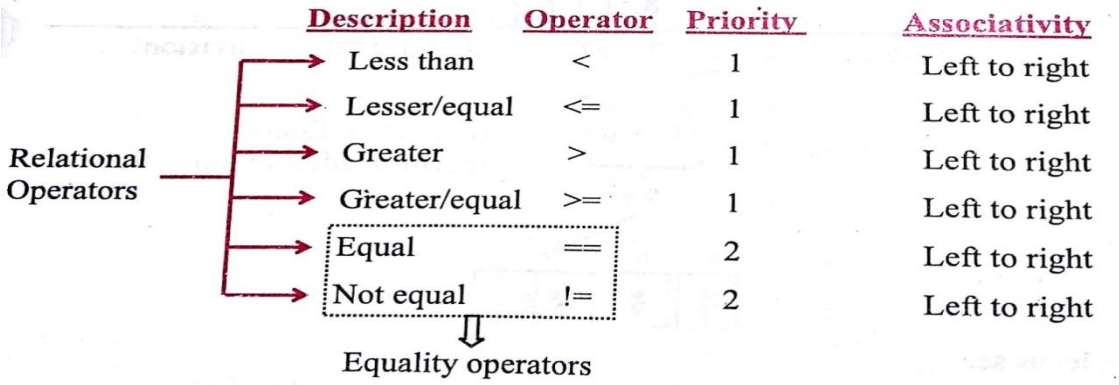
- ✓ **It decrements before (pre) the operand value is used. The operand value is decremented by 1 and this decremented value is used.**

```
Ex:  void main()
     {
        int a=20,b;          // a=20, b?
        b=--a;               // a=--a=a-1= 20-1, b=a=19
          printf("%d",a);// a=19
         printf("%d",b); // b=19
      }
```

iv) Relational operators

- ✓ **The operators that are used to find the relationship between two operands are called relational operators.**
- ✓ **The relationship between the two operand values results in true (always 1) or false (always 0).**

Relational operators available in C are:



- ✓ All the relational operators are having a same priority and left to right associativity.
- ✓ The relational operators have lower precedence than arithmetic operators.

Equality operators

- ✓ C supports two kinds of equality operators to compare their operands for strict equality or inequality.

  equal = =
  not equal ! =

- ✓ Equality operators have lower precedence than the relational operators.

v) Logical operators

- ✓ **The operators that are used to combine two or more relational expressions are called logical operators.**
- ✓ **The output of relational expression is true or false, the output of logical expression is also true or false.**

Logical operators available in C:

Description	Operators	Priority	Associativity
not	!	1	Left to Right
and	&&	2	Left to Right
or	!!	3	Left to Right

Operand1	Operand2	AND(&&)	OR(||)	NOT(!) (Operand1)
True(1)	True(1)	True(1)	True(1)	False(0)
True(1)	False(0)	False(0)	True(1)	False(0)
False(0)	True(1)	False(0)	True(1)	True(1)
False(0)	False(0)	False(0)	False(0)	True(1)

Logical NOT: The logical NOT operator is denoted by '!'. The output of not operator can be true or false. The result is true if the operand value is false and the result is false if the operand is true.

Logical AND: The logical AND operator is denoted by '&&'. The output of and operator is true if both the operands are evaluated to true. If one of the operand is evaluated false, the result is false.

Logical OR: The logical OR operator is denoted by '||'. The output of or operator is true if and only if at least one of the operands is evaluated to true. If both the operands are evaluated to false, the result is false.

Ex: 1. If a, b, c are 3 sides of a triangle, then if a==b && b==c && c==a then triangle is equilateral otherwise not an equilateral triangle.

2. If a, b, c are 3 sides of a triangle then if a==b || b==c || c==a then triangle is isosceles triangle otherwise not a isosceles triangle.

3. int a=10, b:
   b =!a ;
   Output: value of b=0, because a = 10 then !a=0 and !a value assigned to b.

vi) Conditional Operator

- ✓ **The conditional operator is also called as 'ternary operator'.**
- ✓ **An operator that operates on the three operands is called ternary operator.**

Syntax:

```
(expr1)? expr2: expr3;
```

Where,

- expr1 is evaluated first.
- If expr1 is evaluated to true, then expr2is evaluated.
- If expr1 is evaluated to false, then expr3is evaluated.

Example:

1. Write a C program to find the biggest of two numbers using conditional operator.

```
#include<stdio.h>
void main()
{
        int a,b,big;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        big=(a>b)?a:b;
        printf("Big=%d",big);
}
```

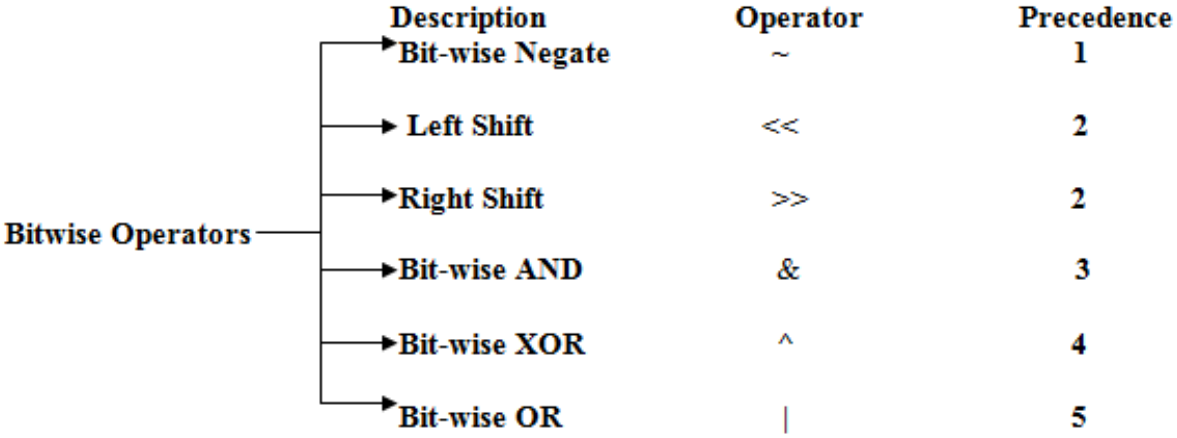
2. Write a C program to find the smallest of two numbers using conditional operator.

```
#include<stdio.h>
void main()
{
        int a,b,small;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        small=(a<b)?a:b;
        printf("Small=%d",big);
}
```

vii) Bitwise Operators

✓ **The operators that are used to manipulate the bits of given data are called bitwise operators.**

Bitwise operators available in C are:



✓ These may only be applied to integral operand. i.e., char, short, int and long whether signed or unsigned.

a. One's Complement(~)

✓ **The operator that is used to change every bit from 0 to 1 and 1 to 0 in the specified operand is called One's complement operator.**

Truth table of One's Complement(~):

Op1	~Op1
0	1
1	0

✓ One's complement operator is denoted by '~(tilde)' symbol.

Ex: Write a C program to show the usage of Bitwise Negate operator.

```
#include<stdio.h>
void main()
{
        int a=10,b;
        b=~a;
        printf("~%d=%d",a,b);
        getch();
}
```

Output: ~10=245

Binary Representation:

10 = 0 0 0 0 1 0 1 0

245= 1 1 1 1 0 1 0 1

b. Left shift operator (<<)

- ✓ **The operator that is used to shift the data by a specified number of bit positions towards left is called 'left shift operator'.**

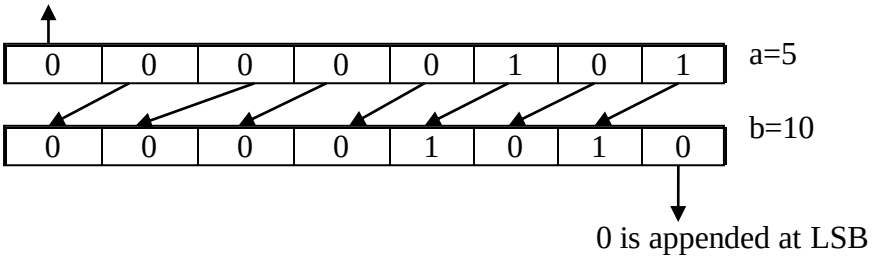
Syntax:

```
b=a<<num;
```

Ex: Write a C program to show the usage of Left Shift operator.

```
#include<stdio.h>
void main()
{
        int a=5,b;
        b=a<<1;
        printf("%d<<1=%d",a,b);
}
Output:  5<<1=10
```

MSB is discarded



c. Right shift operator (>>)

- ✓ **The operator that is used to shift the data by a specified number of bit positions towards right is called 'right shift operator'.**

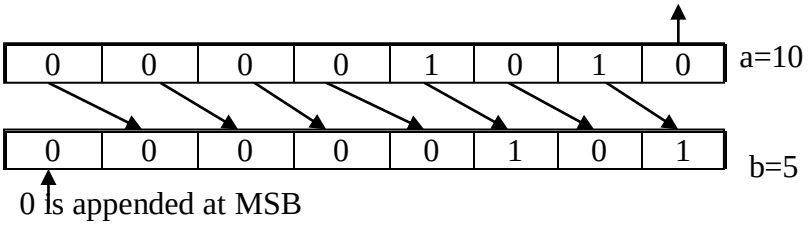
Syntax:

```
b=a>>num;
```

Ex: Write a C program to show the usage of Right Shift operator.

```
#include<stdio.h>
void main()
{
        int a=10,b;
        b=a>>1;
        printf("%d>>1=%d",a,b);
}
Output: 10>>1=5
```

LSB is discarded



d. Bit-wise AND (&)

- ✓ **If the corresponding bit positions in both the operands are 1, then AND operation results in 1, otherwise AND operation results in 0.**

Truth table of Bit-wise AND (&):

Op1	Op2	Op1&Op2
0	0	0
0	1	0
1	0	0
1	1	1

Ex: Write a C program to show the usage of '&' operator.

```
#include<stdio.h>
void main()
{
        int a=10,b=6;
        c=a&b;
        printf("%d&%d=%d",a,b,c);
}
```

Output: 10&6=2

Binary Representation:

```
 10 = 0  0  0  0   1  0  1  0
 06 = 0  0  0  0   0  1  1  0
 02 = 0  0  0  0   0  0  1  0
```

e. Bit-wise OR (|)

- ✓ **If the corresponding bit positions in both the operands are 0, then OR operation results in 0, otherwise OR operation results in 1.**

Truth table of Bit-wise OR (|):

Op1	Op2	Op1|Op2
0	0	0
0	1	1
1	0	1
1	1	1

Ex: Write a C program to show the usage of '|' operator.

```
#include<stdio.h>
void main()
{
        int a=10,b=6;
        c=a|b;
        printf("%d|%d=%d",a,b,c);
}
```

Output: 10|6=14

Binary Representation:

```
 10 = 0  0  0  0   1  0  1  0
 06 = 0  0  0  0   0  1  1  0
 14 = 0  0  0 0   1   1  1  0
```

f. Bit-wise XOR (^)

- ✓ **If the corresponding bit positions in both the operands are different, then XOR operation results in 1, otherwise XOR operation results in 0.**

0^0=0
0^1=1
1^0=1
1^1=0

Ex: Write a C program to show the usage of '^' operator.

```
#include<stdio.h>
void main()
{
        int a=10,b=6;
        c=a^b;
        printf("%d^%d=%d",a,b,c);
}
```

Output: 10^6=12

Binary Representation:

```
10 = 0  0  0  0   1  0  1  0
06 = 0  0  0  0   0  1  1  0
-------------------------------
12 = 0  0  0   0  1  1  0  0
```

viii) Special Operators

- ✓ Comma operator, size of operator and [ ], ->, Indirection operator, *, dot operator etc.

1. Comma operator

- ✓ Comma Operator has the least precedence among all the operators and it is left associative operator.
- ✓ Comma Operator is used in the declaration to separate the variables.
  Ex: int a,b,c;
- ✓ It can be used to separate the items in the list.
  Ex: a=12,345,678;
- ✓ It can be used to combine two or more statements into a single statement.
  Ex: sum=a+b,sub=a-b,mul=a*b,div=a/b,mod=a%b;

2. sizeof()

- ✓ 'sizeof()' operator is used to determine the number of bytes occupied by a variable or a constant in the memory.
- ✓ Ex: sizeof(char) 1 byte
  sizeof(int) 2 bytes
  sizeof(float) 4 bytes

Example program: Write a C program that computes the size of int, float, char and double variables.

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

```
        char ch;
        int x;
        float y;
        double z;
        clrscr();
        printf("Number of bytes occupied by character variable=%d",sizeof(ch));
        printf("Number of bytes occupied by integer variable=%d",sizeof(x));
        printf("Number of bytes occupied by floating-point variable=%d",sizeof(y));
        printf("Number of bytes occupied by double variable=%d",sizeof(z));
}
```

2.1.2 Arithmetic Expressions

- ✓ **The expression consisting of only arithmetic operators such as +, -, *, / and % are called arithmetic expressions.**

Example: Write the equivalent C expression for the Mathematical expressions.

Mathematical Expression	C Equivalent Expression
$S = \frac{a+b+c}{2}$	S=(a + b +c) /2
$area = \sqrt{s(s-a)(s-b)(s-c)}$	area=sqrt(s*(s-a)*(s-b)*(s-c))
$x = \sqrt{2\pi n}$	x=sqrt(2*3.142*n)
$\frac{a}{b}$	a/b
$x = -\frac{b}{2a}$	x=-b/(2*a)
$ax^2 + bx + c$	a*x*x+b*x+c
$\sin\left(\frac{b}{\sqrt{a^2+b^2}}\right)$	sin(b / sqrt(a*a + b*b))
$\tau_1 = \sqrt{\left\{\frac{\sigma_x - \sigma_y}{2}\right\} + \tau xy^2}$	tow1 = sqrt((rowx – rowy)/2 + tow*x*y*y)
$\tau_1 = \sqrt{\left\{\frac{\sigma_x - \sigma_y}{2}\right\}^2 + \tau xy^2}$	tow1 = sqrt(pow((rowx – rowy)/2,2) + tow*x*y*y)
$x1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$	x1 = (-b + sqrt(b*b – 4*a*c)) / (2*a)
$\frac{e^{\lvert a\rvert+b}}{x+y}(2x+3)$	exp(abs(a) + b) / (x + y) * (2*x + 3)
$y = \frac{\alpha+\beta}{Sin\theta} + \lvert x\rvert$	y = (alpha + beta) / sin(theta*3.1416/180) + abs(x)

2.1.3 Associativity of operators

- ✓ When two or more operators have the same precedence, then precedence rules are not applicable.
- ✓ **Associativity determines how the operators with the same precedence are evaluated in an expression.**

The two types of operators Associativity are:

1. **Left Associativity**
2. **Right Associativity**

1. Left Associativity

- ✓ **In an expression, if two or more operators having the same priority are evaluated from left-to-right, then the operators are called Left to Right associative operators.**
- ✓ We normally denote it using L ⟶ R.

What is the result of **8 + 4 + 3?**

In the given expression, the operator "+" is used twice. Since both operators have the same priority, the precedence rules are not applicable. Then the question is *"How to evaluate?"* Normally, we do the evaluation as shown below:

8 + 4 + 3 **[First add 8 and 4 to get 12]**

12 + 3 **[Next add 12 and 3 to get 15]**

15

2. Right Associativity

- ✓ **In an expression, if two or more operators having the same priority are evaluated from right-to-left, then the operators are called Left to Right associative operators.**
- ✓ We normally denote it using R ⟶ L.

Ex: i=j=k=10.

2.1.4 Precedence and order of Evaluation

- ✓ In C language, each operator is associated with priority value.
- ✓ Based on the priority the expressions are evaluated. The priority of each operator is pre-defined in C language.
- ✓ **The order in which the different operators are used to evaluate an expression is called Precedence or hierarchy of operators**
- ✓ The pre-defined priority or precedence order given to each of the operator is called precedence of operator.

Operator category	Operators in order of precedence (highest to lowest)	Associativity
() []	Innermost brackets/function calls Array element reference	L → R (Left to right)
unary operators	++, --, !, sizeof, ~, +, -, &, *	R → L (Right to left)
Member access	* or ->	L → R
arithmetic operators	*, /, %	L → R
arithmetic operators	-, +	L → R
Shift operator	<<, >>	L → R
Relational operators	<, <=, >, >=	L → R
equality operators	==, !=	L → R
Bitwise and	&	L → R
Bitwise xor	^	L → R
Bitwise or	|	L → R
logical and	&&	L → R
logical or	||	L → R
conditional operator	?:	R → L
assignment operator	=, +=, -=, /=, *=, %=	R → L
comma operator	,	L → R

✓ Unary +, - and * have higher precedence than the binary forms.

2.2 Problems on Expression Evaluation

Example 11: Let us evaluate the expression 2 * ((a %5) * (4 + (b-3) / (c+2))) assuming a = 8, b = 15 and c = 4.

The given expression is evaluated as shown below:

```
2 * ( (a % 5) * (4 + (b - 3) / ( c + 2) ) )      Substitute values for a, b and c
2 * ( (8 % 5) * (4 + (15 - 3) / (4 + 2) ) )      Expression inside parentheses
2 * (   3     * (4 + (15 -3) / ( 4 + 2) ) )      Expression inside parentheses
2 * (   3     * (4 +   12    / ( 4 + 2) ) )      Expression inside parentheses
2 * (   3     * (4 +   12    /    6     ) )      Expression inside parentheses
                                                  but division operation
2 * (   3     * (4 +         2          ) )      Expression inside parentheses
2 * (   3     *        6                 )      Expression inside parentheses
2 *             18
36
```

So, the final answer obtained after evaluation = 36

Evaluate the expression 100/20<=10-5+100%10-20==5>=1!=20.

```
100 / 20 <= 10 - 5 + 100 % 10 - 20 == 5 >= 1 != 20    [Division operation]
   5     <= 10 - 5 + 100 % 10 - 20 == 5 >= 1 != 20    [Modulus operation]
   5     <= 10 - 5 +   0      - 20 == 5 >= 1 != 20    [Subtraction operation]
   5     <=    5   +   0      - 20 == 5 >= 1 != 20    [Addition operation]
   5     <=        5           - 20 == 5 >= 1 != 20    [Subtraction operation]
   5     <=                   -15   == 5 >= 1 != 20    [Relational operation]
                0                   == 5 >= 1 != 20    [Relational operation]
                0                   ==   1    != 20    [Equality operation]
                                         0    != 20    [Equality operation]
```

So, result = 1

Evaluate the expression 10 != 10 || 5 < 4 && 8

The above expression is evaluated as shown below:

10 != 10 || 5 < 4 && 8

10 != 10 || 0 && 8

0 || 0 && 8

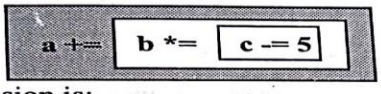
0 || 0

Result = 0

Example 19: Simplify the expression:

a += b *= c -= 5 when a= 1, b = 3 and c = 7

Since the expressions consist of assignment operators and assignment operators are right associative, evaluation is done from right to left as shown below:



The given expression is:

a += b *= c -= 5

a += b *= c -= 5 can be interpreted as shown below

c = c – 5

c = 7 – 5

c = 2

a += b *= 2

a += b *= 2 can be interpreted as

a += b = b*2

b = 3*2

b = 6

a + = 6 can be interpreted as

a = a + 6

a = 1 + 6 = 7

So, **a = 7, b = 6, c = 2**

4: Evaluate the expression a+2 > b && !c || a != d && a-2 <= e where a = 11, b = 6, c = 0, d = 7 and e = 5

Given expression is: a + 2 > b && !c || a != d && a - 2 <= e. After substituting the values, we can evaluate as shown below:

11 + 2 > 6 && !0 || 11 != 7 && 11 - 2 <= 5 [Unary operation performed]

11 + 2 > 6 && 1 || 11 != 7 && 11 - 2 <= 5 [Arithmetic operation performed]

13 > 6 && 1 || 11 != 7 && 11 - 2 <= 5 [Arithmetic operation performed]

13 > 6 && 1 || 11 != 7 && 9 <= 5 [Relational operation performed]

1 && 1 || 11 != 7 && 9 <= 5 [Relational operation performed]

1 && 1 || 11 != 7 && 0 [Relational operation performed]

1 && 1 || 1 && 0 [Logical operation performed]

1 || 1 && 0 [Logical operation performed]

1 || 0 [Logical operation performed]

Result = 1

5: Evaluate the expression a+2 > b || !c && a == d || a-2 <= e
where a = 11, b = 6, c = 0, d = 7 and e = 5

After substituting, the expression can be evaluated as shown below:

11+2 > 6 || !0 && 11 == 7 || 11 - 2 <= 5	[Perform Unary operation]
11+2 > 6 || 1 && 11 == 7 || 11 - 2 <= 5	[Perform Addition operation]
13 > 6 || 1 && 11 == 7 || 11 - 2 <= 5	[Perform Subtraction operation]
13 > 6 || 1 && 11 == 7 || 9 <= 5	[Perform Relational operation]
1 || 1 && 11 == 7 || 9 <= 5	[Perform Relational operation]
1 || 1 && 11 == 7 || 0	[Perform Relational operation]
1 || 1 && 0 || 0	[Perform logical AND operation]
1 || 0 || 0	[Perform logical OR operation]
1 || 0	[Perform logical OR operation]

Result = 1

Evaluate the expression : z = --a * (a+b) / d - c++ * b
where a = 3, b = 4, c = 5 and d = 2.

Solution : z = --a * (a+b) / d - c++ * b

= --a * 7 / d - c++ * b

= --a * 7 / d - 5 * b (Now, a = 3, b = 4, c = 6, d = 2)

= 2 * 7 / d - 5 * b

= 2 * 3 - 5 * b

= 6 - 5 * b

= 6 - 20

∴ z = -14

Evaluate the expression : a += a++ - ++b + a
where, a = 5 & b = 9.

Solution : a += a++ - ++b + a

a = a + (a++ - ++b + a)

a = a + (a++ - 10 + a) (Now, a = 5, b = 10)

a = a + (5 - 10 + a) (Now, a = 6, b = 10)

= a + (5 - 10 + 6)

= a + (-5 + 6)

= a + 1

∴ a = 7

Evaluate the expression: b = (++a) + (a++) + (a++), where a = 6.

Solution: b = (++a) + (a++) + (a++)

b = (++a) + (a++) + 6 (Now, a = 7)

b = (++a) + 7 + 6 (Now, a = 8)

= 9 + 7 + 6 (Now, a = 9)

= 16 + 6

∴ b = 22

2.3 Type Conversion and Typecasting

- ✓ **The process of converting the data or variable from one data type to another data type is called Type Conversion or Typecasting.**
- ✓ **Type conversion is done implicitly by the compiler, whereas typecasting has to be done explicitly by the programmer.**

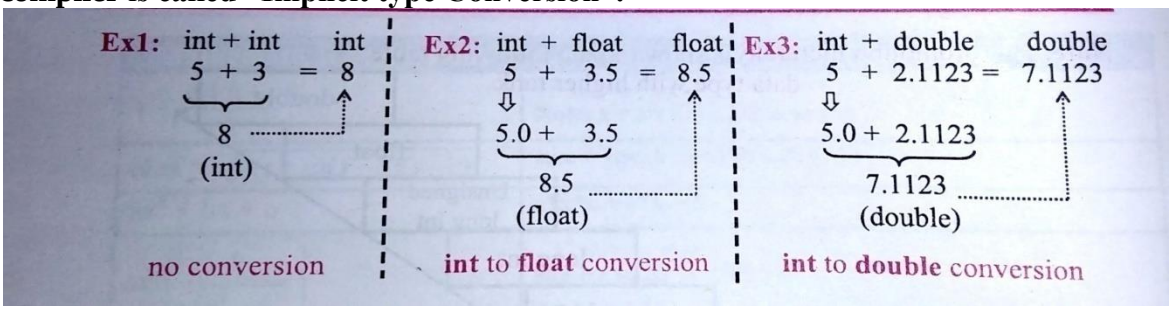
2.3.1 Type Conversion

- ✓ Type conversion is done when the expression has variables of different data types.
- ✓ To evaluate the expression, the **data type is promoted from lower to higher level where the hierarchy of data types (from higher to lower level) can be given as: double, float, long, int, short and char.**

char → short int → int → unsigned int → long int → float → double

Lower Rank Data types → Higher Rank Data types

- ✓ **C compiler converts the data type with lower rank to the data type with higher rank. This process of conversion of data from lower rank to higher rank automatically by the C compiler is called "Implicit type Conversion".**



- ✓ If one operand type is same as that of other operand type, no conversion takes place.

Ex: int + int = int, float + float = float

- ✓ If one operand type is 'int' and other operand type is 'float', then the operand with type int is promoted to 'float' (because float is up in ladder compared with int).
- ✓ Type conversion is automatically done when we assign an integer value to floating point variable. Consider the code given below in which an integer data type is promoted to float. This is known as **promotion (where the lower level data type is promoted to higher type).**

```
float x;
int y=3;
x=y;
```

Now, x=3.0, as automatically integer value is converted into its equivalent floating point representation.

2.3.2 Typecasting

- ✓ Typecasting is also known as **forced conversion.**
- ✓ It is done when the value of a higher data type has to be converted into a value of a lower data type.
- ✓ But this casting is done under the **programmer's control and not under the compiler's control.**
- ✓ **The programmer can instruct the compiler to change the type of the operand or variable from one data type to another data type. This forcible conversion from one data type to another data type is called "Explicit type Conversion" (Type Casting).**

Syntax:

```
(type) Expression
```

Ex: (int) 9.43

i= (int) 5.99 / (int) 2.4, now it becomes 5/2 = 2, i=2

(float) (3/10) = 3.0 / 10.0 = 0.3

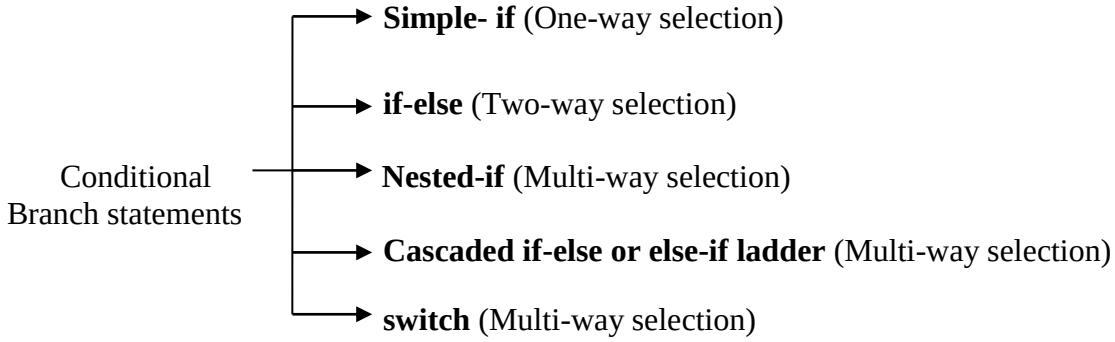
MODULE 2
Decision Control and Looping Statements

2.4 Introduction to Decision Control or Branching Statements

- ✓ **The statements that transfer the control from one place to other place in the program with or without any condition are called branch statements or selection statements.**
- ✓ The branching statements are classified into two types:
  - i. Conditional branch statements
  - ii. Unconditional branch statements

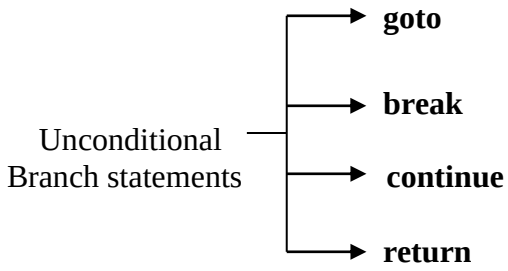
(i) Conditional branch statements

- ✓ **The statements that transfer the control from one place to another place in the program based on some conditions are called Conditional branch statements.**



(ii) Unconditional branch statements

- ✓ **The statements that transfer the control from one place to another place in the program without any condition are called Unconditional branch statements.**



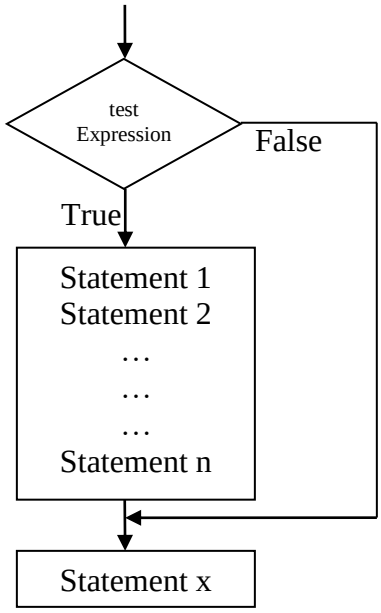
2.5 Conditional Branching Statements

2.5.1 if Statement

- ✓ The 'if' statement is the simplest form of decision control statement.
- ✓ When a set of statements have to be executed when an expression (condition) is evaluated to true or skipped when an expression (condition) is evaluated to false, then if statement is used.
- ✓ It is used whenever there is only one choice (alternative). Hence it is also called as **"One-way decision or selection statement".**

Syntax of if statement

```
if(test Expression)
{
   Statement 1;

   Statement 2;

         ...

         …

         …

   Statement n;

}

Statement x;
```



✓ The ‘if’ structure may include one statement or ‘n’ statements enclosed within curly brackets.

Working Principle

✓ First the test expression is evaluated. If the test expression is true, then the statements of ‘if’ block (statement 1 to n) are executed. Otherwise these statements will be skipped and the execution will jump to statement x.

Rules for if-statement

a) ‘if’ must be followed by an expression and the expression must be enclosed within parenthesis.
b) If multiple statements have to be executed when the expression is true, then all those statements must be enclosed within braces.
c) No semicolon is required for an if-statement.
   if(a<b); // statement indicates a NULL statement. It will ‘do nothing’.

Example Programs for ‘if’ statement

1. Write a C program to determine whether a person is eligible to vote or not.

```
#include<stdio.h>
void main()
{
      int age;
      printf("Enter the age:");
      scanf("%d",&age);
      if(age>=18)
          printf("\nThe person is eligible to vote");
      if(age<18)
          printf("\nThe person is not eligible to vote");
}
```

2. Write a C program to check whether the number is even or odd and print the appropriate message.

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter a number:");
    scanf("%d",&n);
    if(n%2==0)
        printf("Number is even");
    if(n%2=0)
        printf("Number is odd");
}
```

3. Write a C program to print the largest of two numbers.

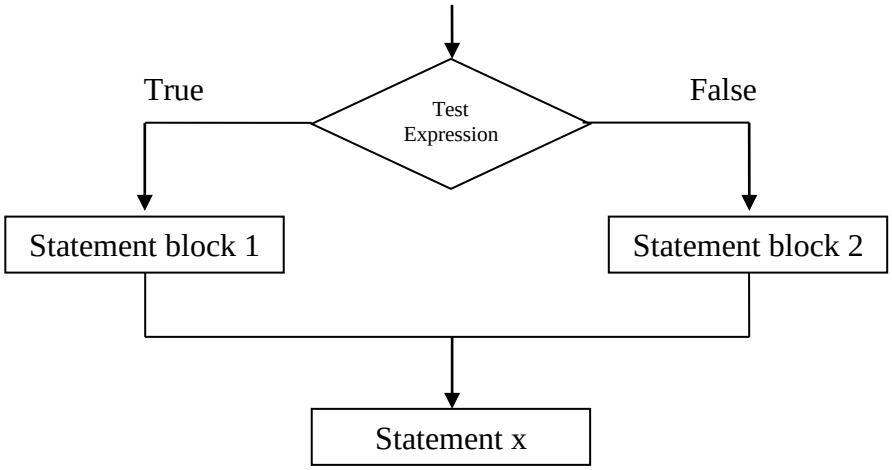
```
#include<stdio.h>
void main()
{
    int a,b;
    printf("Enter the two numbers:");
    scanf("%d%d",&a,&b);
    if(a>b)
        printf("\n a is greater than b");
    if(b>a)
        printf("\n b is greater than a");
}
```

2.5.2 if-else statement

- ✓ If one set of activities have to be performed when an expression is evaluated to true and another set of activities have to be performed when an expression is evaluated to false, then if-else statement is used.

Syntax of if-else statement:

```
if(test Expression)
{
   Statement block 1;

 }

else

{

   Statement block 2;

 }

Statement x;
```



- ✓ The is-else statement is used when we must choose between two choices (alternatives). Hence is also called as **"Two-way Decision or Selection Statement".**

Working Principle

- ✓ According to the if-else construct, first the 'test expression' is evaluated.
- ✓ If the expression is true then Statement block 1 is executed and Statement block 2 is skipped.
- ✓ If the expression is false the Statement block 2 is executed and Statement block 1 is ignored.
- ✓ Now in any case after the Statement block 1 or 2 gets executed the control will pass to Statement x. It is executed in every case.

Example Programs for 'if-else' statement

1. Write a C program to determine whether a person is eligible to vote or not.

```
#include<stdio.h>
 void main()
{
      int age;
       printf("Enter the age:");
      scanf("%d",&age);
      if(age>=18)
          printf("\nThe person is eligible to vote");
      else
          printf("\nThe person is not eligible to vote");
   }
```

2. Write a C program to check whether the number is even or odd and print the appropriate message.

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter a number:");
    scanf("%d",&n);
    if(n%2==0)
        printf("Number is even");
    else
        printf("Number is odd");
}
```

3. Write a C program to enter a character and then determine whether it is a vowel or not.

```
#include<stdio.h>
void main()
{
    char ch;
    printf("Enter any character:");
    scanf("%c",&ch);
    if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u'||ch=='A'||ch=='E'||ch=='I'||ch=='O'||ch=='U')
        printf("\n Character is a vowel");
    else
        printf("\n Character is a consonant");
}
```

4. Write a C program to find whether a given year is leap year or not.

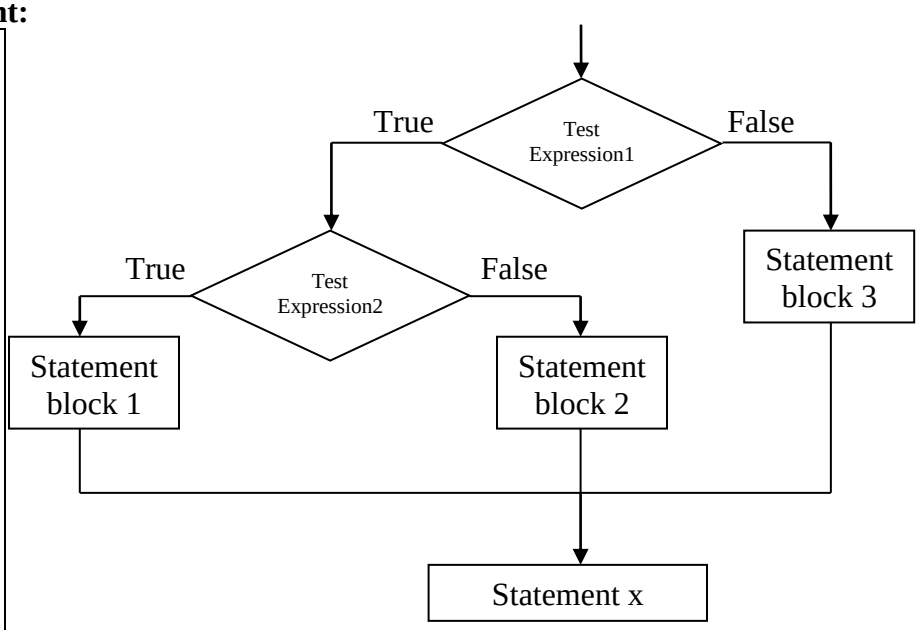
```
#include<stdio.h>
void main()
{
    int year;
    printf("Enter any year:");
    scanf("%d",&year);
    if(((year%4==0)&&(year%100!=0))||(year%400==0))
        printf("%d is a leap year",year);
    else
        printf("%d is not a leap year",year);
}
```

2.5.3 Nested if-else statement

- ✓ **An if-else statement within if statement or an if-else statement within else statement is called "nested if or if-else statement".**
- ✓ When an action has to be performed based on many decisions involving various types of expressions and variables, then this statement is used. So it is called as **"Multi-way decision statement".**

Syntax for nested if-else statement:

```
if(test Expression1)
{
   if(test Expression2)
  {
     Statement block 1;
   }
   else
   {
     Statement block 2;
    }
}
else

{

   Statement block 3;

 }

Statement x;
```



Working Principle

- ✓ If the test expression 1 is evaluated to true, then the test expression 2 is checked for true or false. If the test Expression2 is evaluated to true, then the statements in block 1 are executed, otherwise the statements in block 2 are executed. After executing the inner if-else the control comes out and the statement x is executed.
- ✓ If the test expression1 itself is false, then the statements in block 3 are executed. After executing these statements, the statement x is executed.

Example Programs for 'nested if-else' statement

1. Write a C program to find the largest of three numbers.

```
#include<stdio.h>
void main()
{
      int a,b,c;
      printf("Enter the three numbers:");
      scanf("%d%d%d",&a,&b,&c);
      if(a>b)
      {
```

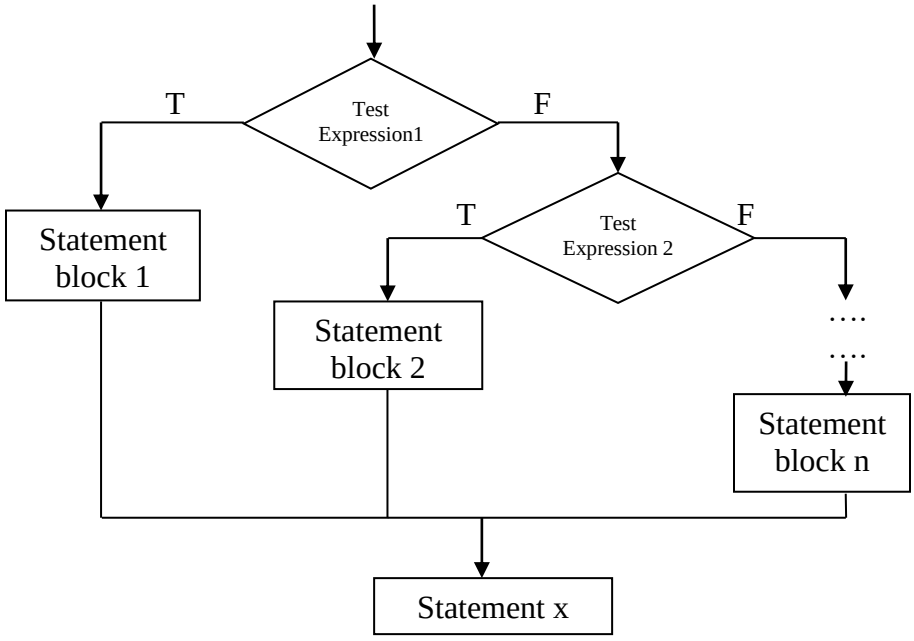
```
        if(a>c)
            printf("Max=%d",a);
        else
            printf("Max=%d",c);
    }
    else
    {
        if(b>c)
            printf("Max=%d",b);
        else
            printf("Max=%d",c);
    }
}
```

2.5.4 Cascaded if-else or if-else-if or else-if-ladder Statement

- ✓ **It is a special case of nested-if statement where nesting takes place only in the else part.**
- ✓ **The orderly nesting of if-else statement only in the else part is called 'else-if-ladder'.**
- ✓ When an action has to be selected based on the range of values, then this statement is used. So it is called **"Multi-way Decision or Selection Statement".**

Syntax :

```
if(test expression1)
{
   Statement block 1;
}
else if(test expression2)
{
   Statement block 2;
}
          ….
          ….
else

{

   Statement block n;

}

Statement x;
```



Working Principle

- ✓ The 'Expressions' are evaluated in order. If any expression is true then the statement associated with it is executed and this terminates the whole chain and statement x is executed.
- ✓ The last 'else' part is executed when all the test expression are false.

Example Programs for 'cascaded if-else' statement

1. Write a C program to demonstrate the use of cascaded if structure.

```
#include<stdio.h>
void main()
{
    int x,y;
    printf("Enter the values of x and y:");
    scanf("%d%d",&x,&y);
    if(x==y)
        printf("\n The two numbers are equal");
    else if(x>y)
        printf("\n %d is greater than %d",x,y);
    else
        printf("\n %d is less than %d",x,y);
 }
```

2. Write a C program to print whether the number is positive, negative or zero.

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter a number:");
    scanf("%d",&n);
    if(n==0)
        printf("Number is zero");
    else if(n>0)
        printf("Number is positive");
    else
        printf("Number is negative");
 }
```

3. Write a C program to input 3 numbers and then find the largest of them using '&&' operator.

```
#include<stdio.h>
void main()
{
    int a,b,c;
    printf("Enter the three numbers:");
    scanf("%d%d%d",&a,&b,&c);
    if(a>b && a>c)
        printf("\n %d is the largest number",a);
    else if(b>a && b>c)
        printf("\n %d is the largest number",b);
    else
        printf("\n %d is the largest number",c);
 }
```

4. Write a C program to display the examination results.

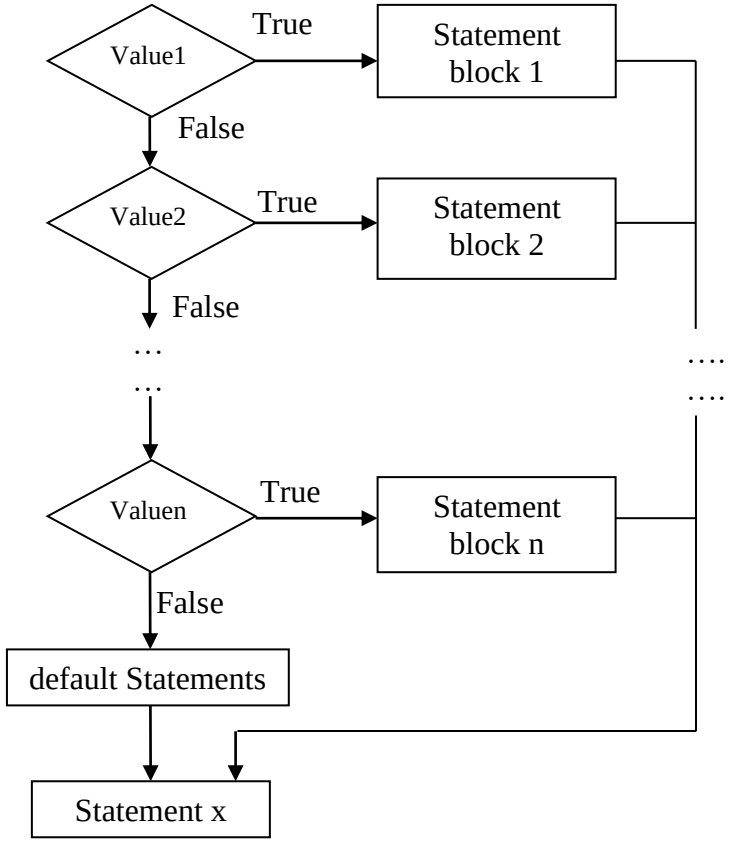
```
#include<stdio.h>
void main()
{
    int marks;
    printf("Enter the marks:");
    scanf("%d",&marks);
    if(marks>=75)
        printf("\n First Class with Distinction");
    else if(marks>=60&&marks<75)
        printf("\n First Class");
    else if(marks>=50&&marks<60)
        printf("\n Second Class");
    else if(marks>=40&&marks<50)
        printf("\n Third Class");
    else
        printf("\n Fail");
  }
```

2.5.5 switch statement

- ✓ The 'switch' statement is a control statement used to select one alternative among several alternatives.
- ✓ It is a **"multi-way decision statement"** that tests whether an expression matches one of the case values and branches accordingly.

switch statement syntax

```
switch(expression)
{
 case value-1: Statement block 1;
               break;
 case value-2: Statement block 2;
               break;
                 …..
                 …..

 case value-n: Statement block n;
               break;
 default:      Statements;

}
Statement x;
```



Working Principle

- ✓ First the expression within switch is evaluated.
- ✓ The value of an expression is compared with all the case values.
- ✓ The value of an expression within switch is compared with the case value-1. If it matches then the statements associated with that case are executed. If not then the case value-2 is compared, if it matches then the associated statements are executed and so on.
- ✓ The "default" statements are executed when no match is found.
- ✓ A default is optional.
- ✓ The 'break' statement causes an exit from the switch. 'break' indicates end of a particular case and causes the control to come out of the switch. **[Significance of break within switch statement]**

Example Programs for 'switch' statement

1. Write a C program to display the grade.

```
#include<stdio.h>
void main()
{  char grade;
   printf("Enter the grade:");
   scanf("%c",&grade);
   switch(grade)
   {
```

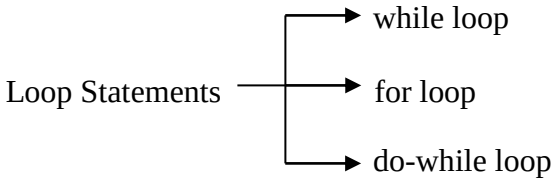
```
        case 'O': printf("Outstanding");
                  break;
        case 'A': printf("Excellent");
                  break;
        case 'B': printf("Good");
                  break;
        case 'C': printf("Fair");
                  break;
        case 'F': printf("Fail");
                  break;
        default : printf("Invalid grade");
      }
     }
```

2. Write a C program to enter a number from 1 to 7 and display the corresponding day of the week using switch.

```
#include<stdio.h>
void main()
{   int day;
    printf("Enter any number:");
    scanf("%d",&day);
    switch(day)
    {
      case 1: printf("Sunday");
              break;
      case 2: printf("Monday");
              break;
      case 3: printf("Tuesday");
              break;
      case 4: printf("Wednseday");
              break;
      case 5: printf("Thursday");
              break;
      case 6: printf("Friday");
              break;
      case 7: printf("Saturday");
              break;
      default : printf("invalid input");
    }
}
```

2.6 Iterative Statements or Loops

- ✓ A set of statements may have to be repeatedly executed for a specified number of times or till a condition is satisfied.
- ✓ **The statements that help us to execute a set of statements repeatedly for a specified number of times or till a condition is satisfied are called as Iterative statements or looping constructs or loop control statements.**
- ✓ These statements are also called as Repetitive or Iterative Statements.

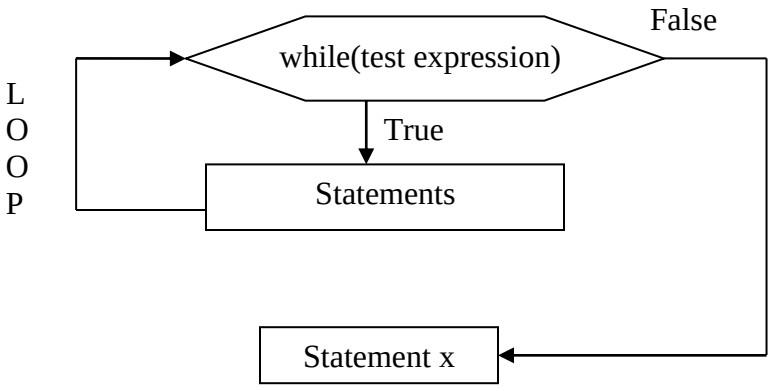


2.6.1 while Loop

- ✓ **It is a control statement using which the programmer can give instructions to the computer to execute a set of statements repeatedly as long as specified condition is satisfied.**
- ✓ The expression is evaluated to TRUE or FALSE in the beginning of the while loop. Hence it is called as **"Entry-Controlled Loop".**

Syntax of while loop

```
while(test expression)
{
   Statements;
}
Statement x;
```



Working principle

- ✓ The test expression is evaluated first, if it is TRUE then the set of statements within the body of the loop are executed repeatedly as long as specified test expression is TRUE.
- ✓ If the test expression is false, then the control comes out of the loop by skipping the execution of the statements within the body of the loop, by transferring the control to the Statement x.

Example programs of 'while' loop

1. Write a C program to print the natural numbers from 1 to 10.

```
#include<stdio.h>
void main()
{
    int i=1;
    while(i<=10)
    {
            printf("%d\n",i);
            i++;
    }
}
```

2. Write a C program to print the natural numbers from 1 to n.

```
#include<stdio.h>
void main()
{
    int i=0,n;
    printf("Enter the value of n:");
    scanf("%d",&n);
    while(i<=n)
    {
            printf("%d\n",i);
            i++;
    }
 }
```

3. Write a C program to print the natural numbers from n to 1.

```
#include<stdio.h>
void main()
{
    int n,i;
    printf("Enter the value of n:");
    scanf("%d",&n);
    i=n;
    while(i>=1)
    {
            printf("%d\n",i);
            i--;
    }
  }
```

4. Write a C program to calculate the sum of the first 10 natural numbers.

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

```
{
    int i=1,sum=0;
    while(i<=10)
    {
            sum=sum+i;
            i++;
    }
     printf("\nSum=%d",sum);
 }
```

5. Write a C program to print the characters from A to Z.

```
#include<stdio.h>
void main()
{
     char ch= 'A';
     while(ch<= 'Z')
     {
          printf("%c\t",ch);
          ch++;
     }
}
```

6. Write a C program to find the factorial of a given number using while statement.

```
#include<stdio.h>
void main()
{
    int fact=1,i=1,n;
    printf("Enter the value of n:\n");
    scanf("%d",&n);
    while(i<=n)
    {
          fact=fact*i;
          i++;
     }
    printf("Factorial of a given number is=%d",fact);
 }
```

7. Write a C program to read a number and determine whether it is palindrome or not.

```
#include<stdio.h>
void main()
{       int digit,rev=0,num,n;
        printf("Enter the number:\n");
        scanf("%d",&num);
        n=num;
        while(num!=0)
        {
```

```
                digit=num%10;
                rev=rev*10+digit;
                num=num/10;
        }
        printf("The reversed number is=%d\n",rev);
        if(n==rev)
                printf(" The number is a palindrome");
        else
                printf("The number is not a palindrome");
}
```

8. Program to find GCD and LCM of 2 numbers.

```
#include<stdio.h>
void main()
{       int a,b,m,n,gcd,lcm,rem;
        printf("Enter the value for m and n :\n");
        scanf("%d %d", &a, &b);
        m=a;
        n=b;
        while(n!=0)
        {
                rem=m%n;
                m = n;
                n = rem;
        }
        gcd = m;
        lcm = ( a * b ) / gcd;
        printf("The GCD of  the  numbers is=%d\n", gcd );
        printf("The LCM of the  numbers is=%d\n", lcm );
}
```

OUT PUT:
Enter the value for m and n:
6
12
The GCD of 6 12 numbers is= 6
The LCM of 6 12 numbers is= 12

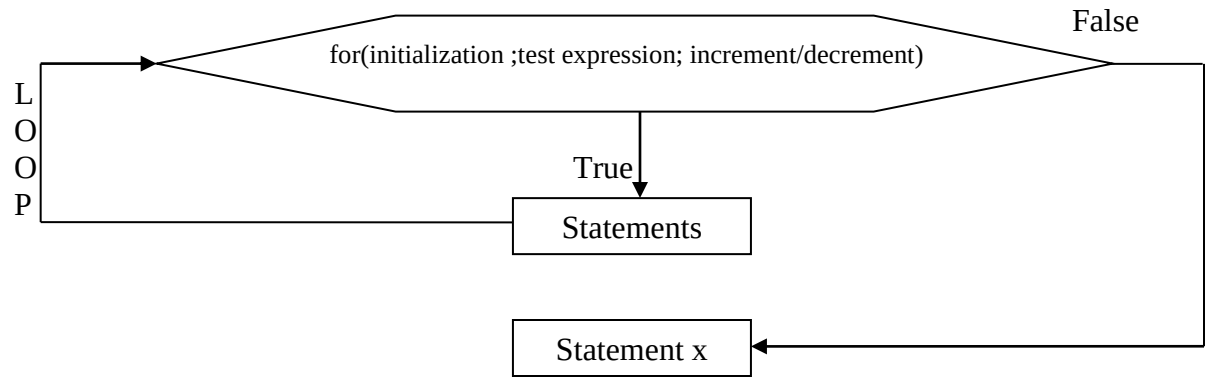
2.6.2 for Loop (Counter- Controlled Loop)

- ✓ **It is a control statement using which the programmer can give instructions to the computer to execute a set of statements repeatedly for a specific number of times.**
- ✓ It is also called **as entry controlled loop** or **pretest loop**.

Syntax of for loop

```
for(initialization ;test expression; increment/decrement)
{
   Statements;
 }
Statement x;
```

Flowchart



Working Principle

- ✓ **initialization:** In this section loop variable is initialized, like i=0, n=0, i=1,n=1. (i and n are loop variables).
- ✓ **test expression:** The test expression may be a relational expression or logical expression or both, which is evaluated to TRUE or FALSE. Depending on the value of the test expression, the body of the loop is executed. If the test expression is TRUE, then the body of the loop is executed. This process of execution of body of the loop is continued as long as the expression is TRUE. When the test expression becomes FALSE, execution of the statements contained in the body of the loop are skipped, thereby transferring the control to the Statement x, which immediately follows the for loop.
- ✓ **increment/decrement:** This section increments or decrements the loop variables after executing the body of the loop.

Infinite Loop

- ✓ A for loop without a test expression is an Infinite loop.

Ex: for(i=0; ;i++)
{
……
}

is an "infinite" loop.

Example programs of 'for' loop

1. Write a C program to print the natural numbers from 1 to 10.

```
#include<stdio.h>
void main()
{
        int i;
        for(i=1;i<=10;i++)
        {
                printf("%d\n",i);
        }
}
```

2. Write a C program to print the natural numbers from 1 to n.

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

3. Write a C program to compute the sum of the series 1+2+3+……+n.

```
#include<stdio.h>
void main()
{
        int i,sum=0,n;
        clrscr();
        printf("Enter the number of terms:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                sum=sum+i;
        }
        printf("\nSum of the series=%d",sum);
  }
```

4. Write a C program to generate 'n' Fibonacci numbers.

```
#include<stdio.h>
void main()
{
        int n,f1,f2,f3,i;
        f1=0;
        f2=1;
        printf("Enter the value of n:");
        scanf("%d",&n);
        if(n==1)
                printf("%d",f1);
        else
        {
                printf("%d\t%d\t",f1,f2);
                for(i=3;i<=n;i++)
                {
                        f3=f1+f2;
                        printf("%d\t",f3);
                        f1=f2;
                        f2=f3;
```

```
                    }
                }
            }
```

5. Write a C program to print the characters from A to Z.

```
#include<stdio.h>
void main()
{
        char ch;
        for(ch= 'A';ch<= 'Z';ch++)
        {

                printf("%c\t",ch);
        }

}
```

Output: A B C D E F ----------Z

6. Write a C program to find the factorial of a given number using for statement.

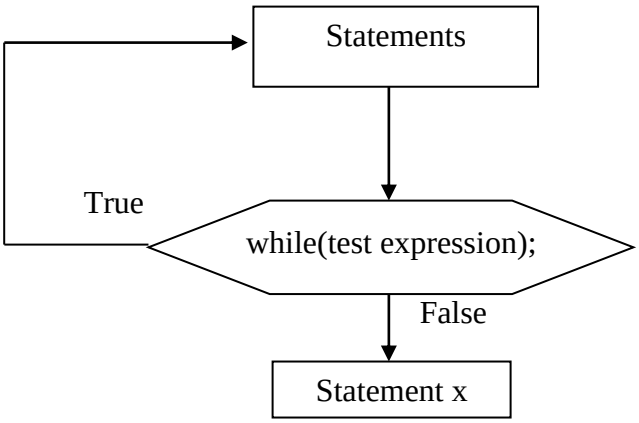
```
 #include<stdio.h>
void main()
{
        int fact=1,i,n;
        printf("Enter the value of n:\n");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                fact=fact*i;
         }
        printf("Factorial of a given number is=%d",fact);

 }
```

2.6.3 do-while Loop

- ✓ **do- while loop is used when a set of statements have to be repeatedly executed at least once.**
- ✓ Since the test expression is evaluated to TRUE or FALSE at the end of do-while loop, the do-while loop is called as **exit-controlled loop**.
- ✓ The while and for loop test the expression at the top.
- ✓ The do-while tests the expression at the bottom after making each passes through the loop body.

Syntax of do-while loop

```
do{
        Statements;

   }while(test expression);

Statement x;
```



Working Principle

- ✓ It is a **post-test or bottom-testing loop** and hence the statements contained within the body of the loop are executed first and then the expression is evaluated to TRUE or FALSE. If it is TRUE, then the statements contained within the body of the loop are executed once again and the expression is evaluated. This is repeated until the expression is evaluated to FALSE.
- ✓ If the expression is FALSE, then the control comes out of the loop by skipping the execution of the statements within the body of the loop, by transferring the control to the Statement x.

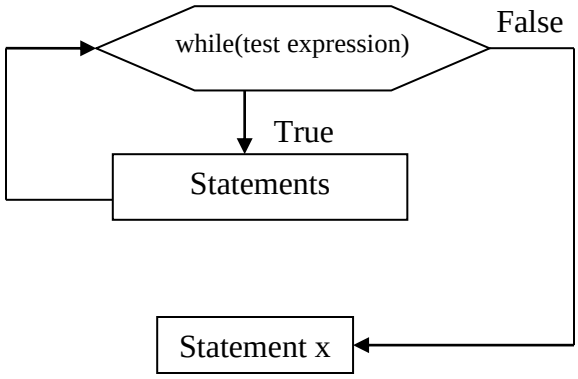
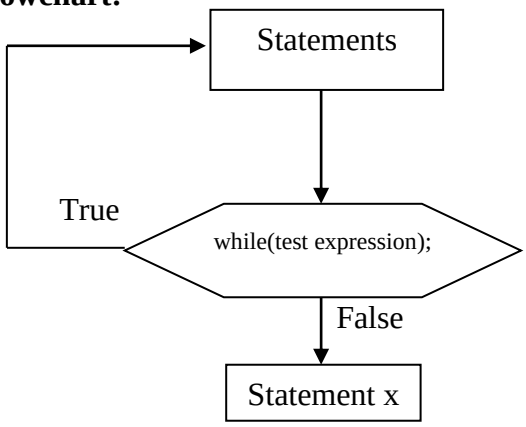
2.6.4 Difference between while loop and do-while loop

	while loop	**do-while loop**
1	It is **entry controlled loop. (top-testing)**	It is **exit controlled loop**. **(bottom-testing)**
2	It is pre-test loop.	It is post-test loop.

3	**Syntax:**
	while(expression)
	{
	Statements;
	}
		Syntax:
		do {
		Statements;
		} while(expression);

4	**Working Principle:** If the expression is evaluated to true, then the statements within the body of loop are executed. If the expression is false, then the statements within the body of the loop are not executed.	**Working Principle:** On reaching do statement, it proceeds to execute the statements within the body of the loop irrespective of the test expression. If the expression is evaluated to true, then the statements within the body of loop are executed once again. If the expression is false, then the statements within the body of the loop are not executed.
5	In while loop, while statement does not end with semicolon (;).	In do-while loop, while statement ends with semicolon (;).

6	Ex:
	int i=0,sum=0;
	while(i<=n)
	{
	sum=sum+i;
	i=i+1;
	}
		Ex:
		int i=0,sum=0;
		do{
		sum=sum+i;
		i=i+1;
		}while(i<=n);

7	**Flowchart:** 	**Flowchart:** 

Example programs of 'do-while' loop

1. Write a C program to print the natural numbers from 1 to n.

```
#include<stdio.h>
void main()
{
    int i=1,n;
    printf("Enter the value of n:");
    scanf("%d",&n);
    do
    {
            printf("%d\n",i);
            i++;
    }while(i<=n);
}
```

2. Write a C program to calculate the sum and average of first 'n' numbers.

```
    #include<stdio.h>
    void main()
    {
            int n,i=0,sum=0;
            float avg=0.0;
            printf("Enter the value of n:");
            scanf("%d",&n);
           do
           {
                    sum=sum+i;
                    i++;

           }while(i<=n);
           avg=sum/n;
           printf("The sum of numbers=%d",sum);
           printf("The average of numbers=%f",avg);
    }
```

2.7 Nested Loops

- ✓ The loops that can be placed inside other loops.
- ✓ It will work with any loops such as while, do-while and for.
- ✓ Nested loop is commonly used with the for loop because this is easiest to control.
- ✓ The 'inner for loop' can be used to control the number of times that a particular set of statements will be executed.
- ✓ The 'outer for loop' can be used to control the number of times that a whole loop is repeated.

Example Program: Write a C program to print the following output.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4
```

```
#include<stdio.h>
void main( )
{
        int i,j;
        for(i=1;i<=5;i++)
        {
                printf("\n");
                for(j=1;j<=i;j++)
                {
                        printf("%d",j);
                }
        }
}
```

Outer for loop

Inner for loop

2.8 Unconditional Branch statements

2.8.1 break statement

- ✓ **In C, break statement is used to terminate the execution of the nearest enclosing loop in which it appears.**
- ✓ It is 'jump' statement which can be used in **switch statement and loops**.
- ✓ **The break statement in switch statement causes the control to come out of the switch statement and transfers the control to the statement which follows the switch statement.**

Ex:

```
switch(ch)
{
 case 1: printf("1st statement");
         break;
 case 2: printf("2nd  statement");
         break;
 default: printf("nth statement");
}
```

If 'case 1' is selected by the programmer then the output will be 1st statement only. It will directly come out of the switch.

- ✓ **If break is executed in a loop (for/while/do-while) then the control comes out of the loop and the statement following the loop will be executed.**

Syntax

1. while(…)
   {
   ……
   if(condition)
   break;
   ……
   }
   ……

   Transfers the control out of the while loop

2. do
   {
   ……
   if(condition)
   break;
   ……
   }while(condition);
   ……

   Transfers the control out of the do-while loop

3. for(…)
   {
   ……
   if(condition)
   break;
   ……
   }
   ……

   Transfers the control out of the for loop

Example programs of 'break' statement

1. Write a C program to print the numbers (0 to 4) using break.

```
#include<stdio.h>
void main()
{
        int i=0;
        while(i<=10)
        {
                if(i==5)
                    break;
                printf("%d\n",i);
                i=i+1;
        }
}
```

2.8.2 continue statement

- ✓ During execution of a loop, it may be necessary to skip a part of the loop based on some condition. In such cases, we use continue statement.
- ✓ **The continue statement is used only in the loops to terminate the current iteration and continue with the remaining iterations.**

Syntax:

1. while(…)
   {
   if(condition)
   continue;
   …….
   }
   …….

   Transfers the control to the expression of the while loop

2. do
   {
   ……
   if(condition)
   continue;
   ……
   }while(condition);
   ……

   Transfers the control to the expression of the do-while loop

3. for(…)
   {
   ……
   if(condition)
   continue;
   ……
   }
   ……

   Transfers the control to the expression of the for loop

Example programs of 'continue' statement

1. Write a C program to display the output in the following form 1 3 4 5

```
#include<stdio.h>
void main()
{
        int i;
        for(i=1;i<=5;i++)
        {
                if(i==2)
                    continue;
                printf("%d",i);
         }

}
```

Output: 1 3 4 5 [if i==2 then the continue statement is executed and the statements following continue are skipped]

2.8.3 goto statement

- ✓ **The goto statement is a 'jump' statement that transfers the control to the specified statement (Label) in a program unconditionally.**
- ✓ The specified statement is identified by 'label'(symbolic name). Label can be any valid variable name that is followed by a colon (:).

Syntax

```
goto label;
    ……
    …….
label:
    Statements
```
Forward Jump

```
label:
   Statements
    ……
    ……
goto label;
```
Backward Jump

Example programs of 'goto' statement

1. Write a C program to calculate the sum of all numbers entered by the user.

```
#include<stdio.h>
void main()
{
        int n,i,sum=0;
        printf("Enter the number:\n");
        scanf("%d",&n);
        sum=i=0;
        top:    sum=sum+n;
                i=i+1;
       if(i<=n)         goto top;
                printf("Sum of Series=%d",sum);

 }
```

2.9 Finding Roots of Quadratic Equations

- ✓ Quadratic Equations $ax^2+bx+c=0$.
- ✓ Formula for calculating Roots of Quadratic equations. $\boldsymbol{x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}}$

```
#include<stdio.h>
#include<math.h>
void main()
{
```

```
        float a,b,c,disc,root1,root2,realp,imgp;
        printf("enter the co-efficients\n");
        scanf("%f%f%f",&a,&b,&c);
        if(a==0||b==0||c==0)
        {
                printf("invaled input");
                exit(0);
        }
        disc=b*b-4*a*c;
                printf("discriminate=%f\n",disc);
        if(disc==0)
        {
                root1=root2=-b/(2*a);
                printf("roots are equal\n");
                printf("root1=%f\n root2=%f\n",root1,root2);
        }
        else if(disc>0)
        {
                root1=(-b+sqrt(d))/(2*a);
                root2=(-b-sqrt(d))/(2*a);
                printf("roots are distinct\n");
                printf("root1=%f\n root2=%f\n",root1,root2);
        }
        else
        {
                printf("roots are imaginary\n");
                realp=-b/(2*a);
                imgp=sqrt(fabs(d))/(2*a);
                printf("root1=%f+i%f\n",realp,imgp);
                printf("root2=%f-i%f\n",realp,imgp);
        }
    }
}
```

2.10 Write a C Program to display the following by reading the number of rows as input,

```
                1
            1   2   1
        1   2   3   2   1
    1   2   3   4   3   2   1
    ---------------------------
    nth row
```

```
#include <stdio.h>
void main()
{
        int i,j,n;
        printf("Input number of rows : ");
        scanf("%d",&n);
        for(i=0;i<=n;i++)
        {
                /*print blank spaces */
                for(j=1;j<=n-i;j++)
                        printf(" ");
                /* Display number in ascending order upto middle*/
                for(j=1;j<=i;j++)
                        printf("%d",j);
                /* Display number in reverse order after middle */
                for(j=i-1;j>=1;j--)
                        printf("%d",j);
                printf("\n");
        }
}
```

2.11 Computation of Binomial Coefficients

- ✓ The binomial coefficient C(n,r) or nCr is the number of ways of picking 'r' unordered outcomes from 'n' possibilities, also known as a combination or combinatorial number. The number of ways that r objects can be chosen from among n objects;

$$^{n}C_{r} = n\ !\ /(n - r)!\ *\ r\ !$$

Ex: C Program to Calculate Binomial coefficient.

```
#include<stdio.h>
void main()
{
        int i,n,r;
        long int x,y,z,nCr;
        printf("enter the value of n and r\n");
        scanf("%d %d", &n, &r);
```

```
        x = y = z = 1;
        for(i = n; i >=1; i--)
        {
                x = x * i;
        }
        for(i = n-r; i >=1; i--)
        {
                y = y * i;
        }
        for(i = r; i >=1; i--)
        {
                z = z * i;
        }
        nCr = x / (y * z);
        printf("%d C %d= %d", n ,r, nCr);
}
```

Output:

Enter the value of n and r:
6
2
6C2 = 15

MODULE 3
Arrays

3.1 Introduction

- ✓ **An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).**
- ✓ **The subscript is an ordinal number which is used to identify an element of the array.**

3.2 Declaration of Arrays

- ✓ An array must be declared before being used.
- ✓ Arrays are declared using the following **syntax:**

  type name[size];
- ✓ Declaring an array means specifying the following:
  - **Data type**—the kind of values it can store, for example, int, char, float, double, or any other valid data type..
  - **Name**—to identify the array.
  - **Size**—the maximum number of values that the array can hold. i.e., the maximum number of elements that can be stored in the array.

  For example, if we write,

  int marks[10];

  then the statement declares marks to be an array containing 10 elements. In C, the array index starts from zero. The first element will be stored in marks[0], second element in marks[1], and so on. Therefore, the last element, that is the 10th element, will be stored in marks[9]. Note that 0, 1, 2, 3 written within square brackets are the subscripts. In the memory, the array will be stored as shown in Fig. 3.2.

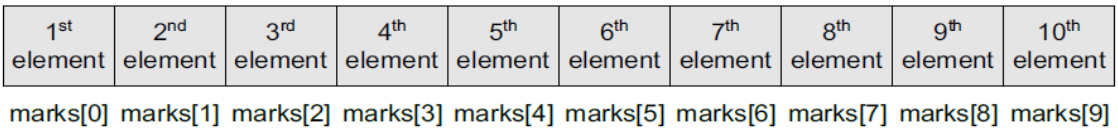


Figure 3.2 Memory representation of an array of 10 elements

- ✓ Figure 3.3 shows how different types of arrays are declared.

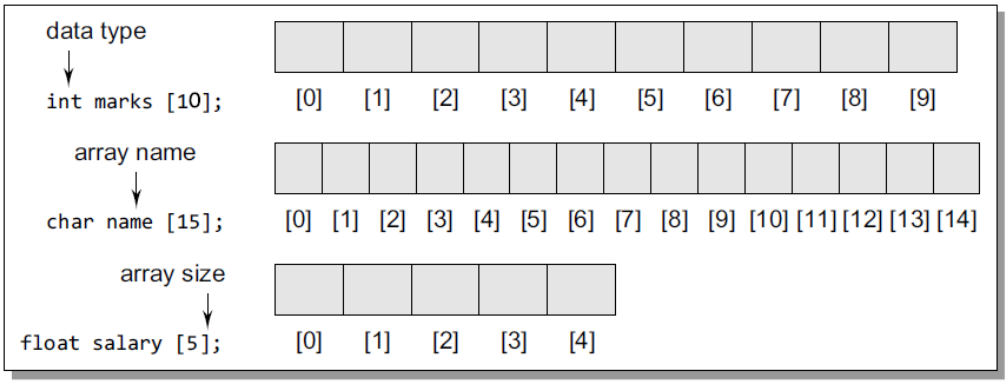


Figure 3.3 Declaring arrays of different data types and sizes

3.3 Accessing the Elements of an Array

- ✓ To access all the elements, **we must use a loop.** That is, **we can access all the elements of an array by varying the value of the subscript into the array.**
- ✓ But note that the subscript must be an integral value or an expression that evaluates to an integral value.
- ✓ As shown in Fig. 3.2, the first element of the array marks[10] can be accessed by writing marks[0]. Now to process all the elements of the array, we use a loop as shown in Fig. 3.4.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
        marks[i] = -1;
```

Figure 3.4 Code to initialize each element of the array to –1

- ✓ Figure 3.5 shows the result of the code shown in Fig. 3.4. The code accesses every individual element of the array and sets its value to –1. In the for loop, first the value of marks[0] is set to –1, then the value of the index (i) is incremented and the next value, that is, marks[1] is set to –1. The procedure continues until all the 10 elements of the array are set to –1.

– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

3.3.1 Calculating the Address of Array Elements

- ✓ The **array name** is a symbolic reference to the **address of the first byte of the array.**
- ✓ When we **use the array name**, we are actually referring to the **first byte of the array.**
- ✓ The **subscript or the index** represents the **offset from the beginning of the array to the element being referenced.**
- ✓ That is, with just the **array name and the index**, **C can calculate the address of any element in the array.**
- ✓ Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient.
- ✓ The address of other data elements can simply be calculated using the base address. The formula to perform this calculation is,

Address of data element, A[k] = BA(A) + w(k – lower_bound)

Here, A is the array,

k is the index of the element of which we have to calculate the address,

BA is the base address of the array A, and

w is the size of one element in memory, for example, size of int is 2.

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
marks[4] = 1000 + 2(4 – 0)
         = 1000 + 2(4) = 1008
```

3.3.2 Calculating the Length of an Array

- ✓ The length of an array is given by the number of elements stored in it.
- ✓ The general formula to calculate the length of an array is

Length = upper_bound – lower_bound + 1

where, upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

Example 3.2 Let `Age[5]` be an array of integers such that

`Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7`

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

`Length = upper_bound - lower_bound + 1`

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 - 0 + 1 = 5`

3.4 Storing Values in Arrays

- ✓ When we declare an array, we are just allocating space for its elements; no values are stored in the array.
- ✓ There are three ways to store values in an array. First, to **initialize the array elements during declaration**; second, to **input values for individual elements from the keyboard**; third, to **assign values to individual elements**.
- ✓ This is shown in Fig. 3.6.

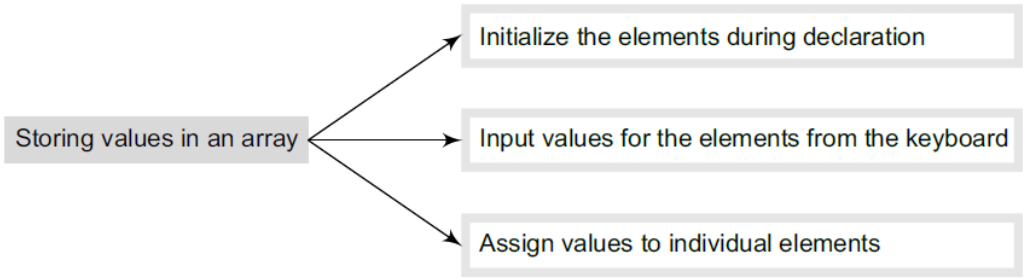


Figure 3.6 Storing values in an array

1. Initializing Arrays during Declaration

- ✓ The elements of an array can be initialized at the **time of declaration**, just as **any other variable.**
- ✓ When an array is initialized, we need **to provide a value for every element in the array.**
- ✓ Arrays are initialized by writing,

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

- ✓ Note that the values are written within curly brackets and every value is separated by a comma. It is a compiler error to specify more values than there are elements in the array. When we write,

int marks[5]={90, 82, 78, 95, 88};

✓ An array with the name marks is declared that has enough space to store five elements. The first element, that is, marks[0] is assigned value 90. Similarly, the second element of the array, that is marks[1], is assigned 82, and so on. This is shown in Fig. 3.7.

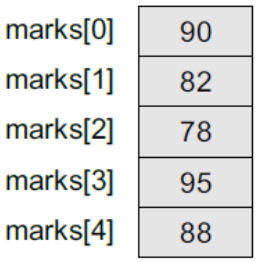


Figure 3.7 Initialization of array `marks[5]`

✓ While initializing the array at the time of declaration, the programmer may omit the size of the array. For example,

int marks[ ]= {98, 97, 90};

The above statement is absolutely legal. Here, the compiler will allocate enough space for all the initialized elements. Note that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros. Figure 3.8 shows the initialization of arrays.

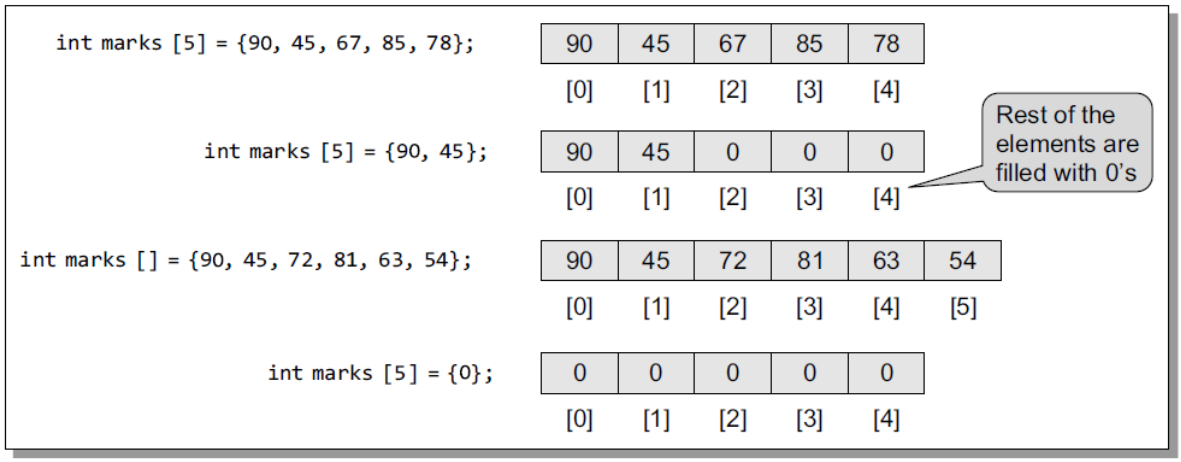


Figure 3.8 Initialization of array elements

2. Inputting Values from the Keyboard

✓ An array can be initialized by **inputting values from the keyboard.**

✓ In this method, a **while/do–while or a for loop** is executed to input the value for each element of the array. For example, look at the code shown in Fig. 3.9.

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

Figure 3.9 Code for inputting each element of the array

✓ In the code, we start at the index i at 0 and input the value for the first element of the array. Since the array has 10 elements, we must input values for elements whose index varies from 0 to 9.

3. Assigning Values to Individual Elements

- ✓ The third way is to **assign values to individual elements of the array by using the assignment operator.**
- ✓ Any value that evaluates to the data type as that of the array can be assigned to the individual array element.
- ✓ A simple assignment statement can be written as

  marks[3] = 100;

  Here, 100 is assigned to the fourth element of the array which is specified as marks[3].
- ✓ To copy an array, you must copy the value of every element of the first array into the elements of the second array. Figure 3.10 illustrates the code to copy an array. In Fig. 3.10, the loop accesses each element of the first array and simultaneously assigns its value to the corresponding element of the second array. The index value i is incremented to access the next element in succession. Therefore, when this code is executed, arr2[0] = arr1[0], arr2[1] = arr1[1], arr2[2] = arr1[2], and so on.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
       arr2[i] = arr1[i];
```

Figure 3.10 Code to copy an array at the individual element level

- ✓ For example, if we want to fill an array with even integers (starting from 0), then we will write the code as shown in Fig. 3.11. In the code, we assign to each element a value equal to twice of its index, where the index starts from 0. So after executing this code, we will have arr[0] = 0, arr[1] = 2, arr[2] = 4, and so on.

```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
       arr[i] = i*2;
```

Figure 3.11 Code for filling an array with even numbers

3.5 Operations on Arrays

- ✓ There are a number of operations that can be performed on arrays.
- ✓ These operations include:
  - **Traversing an array**
  - **Inserting an element in an array**
  - **Searching an element in an array**
  - **Deleting an element from an array**
  - **Merging two arrays**
  - **Sorting an array in ascending or descending order**

3.5.1 Traversing an Array

- ✓ Traversing an array means **accessing each and every element of the array for a specific purpose.**
- ✓ Traversing the data elements of an array A can include **printing every element, counting the total number of elements, or performing any process on these elements**.

✓ The algorithm for array traversal is given in Fig. 3.12.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:       Apply Process to A[I]
Step 4:       SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

✓ In Step 1, we initialize the index to the lower bound of the array. In Step 2, a while loop is executed. Step 3 processes the individual array element as specified by the array name and index value. Step 4 increments the index value so that the next array element could be processed. The while loop in Step 2 is executed until all the elements in the array are processed, i.e., until I is less than or equal to the upper bound of the array.

Examples:

Write a program to read and display n numbers using an array.

```
#include <stdio.h>
void main()
{
        int i, n, a[10];
        printf("Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("Enter the array elements:\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("The array elements are:\n ");
        for(i=0;i<n;i++)
                printf("%d\n", a[i]);
}
```

Output

```
Enter the number of elements in the array: 5
Enter the array elements:
1
2
3
4
5
The array elements are:
1
2
3
4
5
```

3.5.2 Inserting an Element in an Array

✓ **If an element has to be inserted at the end of an existing array, then we just have to add 1 to the upper_bound and assign the value.** Here, we assume that the memory space allocated for the array

is still available. For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

- ✓ Figure 3.13 shows an algorithm to insert a new element to the end of an array. In Step 1, we increment the value of the upper_bound. In Step 2, the new value is stored at the position pointed by the upper_bound.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Figure 3.13 Algorithm to append a new element to an existing array

- ✓ For example, let us assume an array has been declared as

```
int marks[60];
```

The array is declared to store the marks of all the students in a class. Now, suppose there are 54 students and a new student comes and is asked to take the same test. The marks of this new student would be stored in marks[55]. Assuming that the student secured 68 marks, we will assign the value as

```
marks[55] = 68;
```

- ✓ Consider an array whose elements are arranged in ascending order. Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array. To do this, we must first find the location where the new element will be inserted and then move all the elements (that have a value greater than that of the new element) one position to the right so that space can be created to store the new value.

Example 3.3 `Data[]` is an array that is declared as `int Data[20];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) Calculate the length of the array.
(b) Find the `upper_bound` and `lower_bound`.
(c) Show the memory representation of the array.
(d) If a new data element with the value 75 has to be inserted, find its position.
(e) Insert a new data element 75 and show the memory representation after the insertion.

Solution

(a) Length of the array = number of elements
Therefore, `length of the array = 10`

(b) By default, `lower_bound = 0` and `upper_bound = 9`

(c)

12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one position towards the right to accommodate the new value

(e)

12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

Algorithm to Insert an Element in the Middle of an Array

✓ The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted
(b) N, the number of elements in the array
(c) POS, the position at which the element has to be inserted
(d) VAL, the value that has to be inserted

✓ In the algorithm given in Fig. 3.14, in Step 1, we first initialize I with the total number of elements in the array. In Step 2, a while loop is executed which will move all the elements having an index greater than POS one position towards right to create space for the new element. In Step 5, we increment the total number of elements in the array by 1 and finally in Step 6, the new value is inserted at the desired position.

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:          SET A[I + 1] = A[I]
Step 4:          SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

Initial `Data[]` is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling `INSERT(Data, 6, 3, 100)` will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

3.5.3 Deleting an Element from an Array

✓ **Deleting an element from an array means removing a data element from an already existing array.**

✓ **If the element has to be deleted from the end of the existing array, then we just have to subtract 1 from the upper_bound.**

✓ Figure 3.15 shows an algorithm to delete an element from the end of an array.

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Figure 3.15 Algorithm to delete the last element of an array

✓ For example, if we have an array that is declared as

int marks[60];

The array is declared to store the marks of all the students in the class. Now, suppose there are 54 students and the student with roll number 54 leaves the course. The score of this student was stored in marks[54]. We just have to decrement the upper_bound. Subtracting 1 from the upper_bound will indicate that there are 53 valid data in the array.

✓ Consider an array whose elements are arranged in ascending order. Now, suppose an element has to be deleted, probably from somewhere in the middle of the array. To do this, we must first find the location from where the element has to be deleted and then move all the elements (having a value greater than that of the element) one position towards left so that the space vacated by the deleted element can be occupied by rest of the elements.

Example 3.4 `Data[]` is an array that is declared as `int Data[10];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) If a data element with value 56 has to be deleted, find its position.

(b) Delete the data element 56 and show the memory representation after the deletion.

Solution

(a) Since the elements of the array are stored in ascending order, we will compare the value that has to be deleted with the value of every element in the array. As soon as `VAL = Data[I]`, where `I` is the index or subscript of the array, we will get the position from which the element has to be deleted. For example, if we see this array, here `VAL = 56`. `Data[0] = 12` which is not equal to 56. We will continue to compare and finally get the value of `POS = 4`.

(b)

12	23	34	45	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]

Algorithm to delete an element from the middle of an array

✓ The algorithm DELETE will be declared as DELETE(A, N, POS). The arguments are:

(a) A, the array from which the element has to be deleted

(b) N, the number of elements in the array

(c) POS, the position from which the element has to be deleted

✓ Figure 3.16 shows the algorithm in which we first initialize I with the position from which the element has to be deleted. In Step 2, a while loop is executed which will move all the elements having an index greater than POS one space towards left to occupy the space vacated by the deleted element. In Step 5, we decrement the total number of elements in the array by 1.

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:          SET A[I] = A[I + 1]
Step 4:          SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Figure 3.16 Algorithm to delete an element from the middle of an array

✓ Calling DELETE (Data, 6, 2) will lead to the following processing in the array.

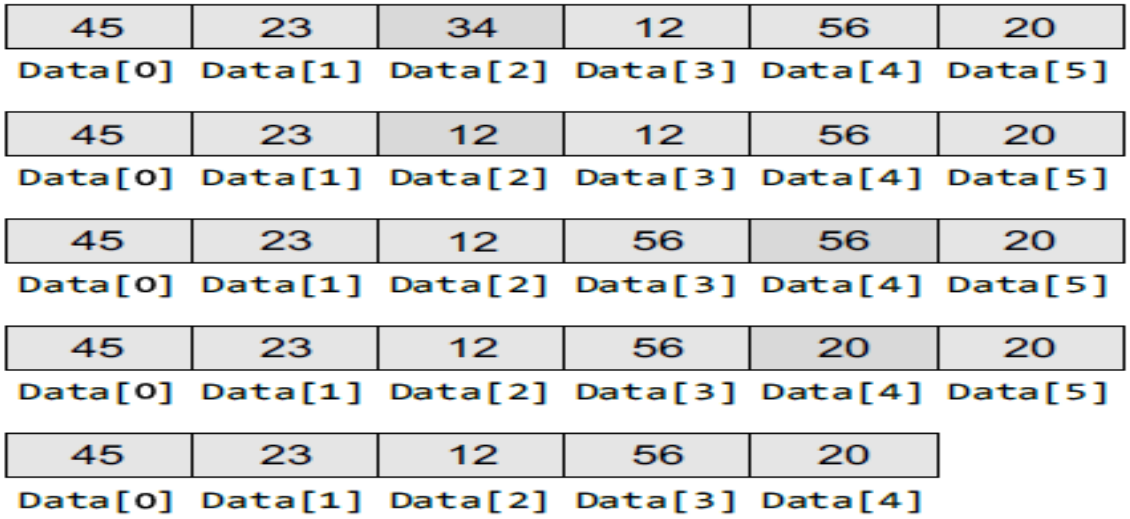


Figure 3.17 Deleting elements from an array

3.5.4 Merging Two Arrays

✓ **Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.** This operation is shown in Fig 3.18.

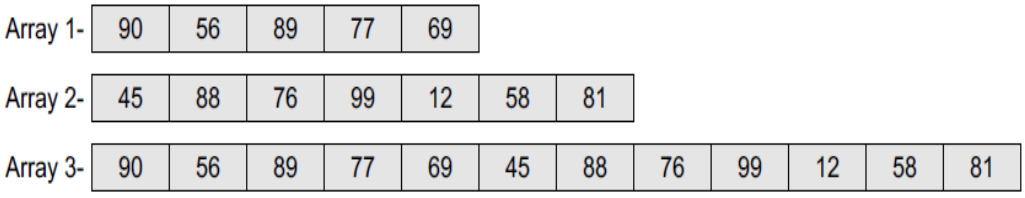


Figure 3.18 Merging of two unsorted arrays

✓ If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult. The task of merging can be explained using Fig. 3.19.

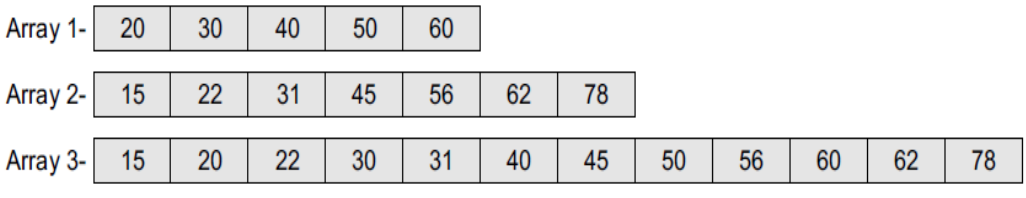


Figure 3.19 Merging of two sorted arrays

3.5.5 Searching for a Value in an Array

✓ **Searching means to find whether a particular value is present in an array or not.**

✓ If the **value is present** in the array, then **searching is said to be successful** and the searching process gives the **location of that value in the array**. However, if the **value is not present** in the array, the

searching process displays an **appropriate message** and in this case **searching is said to be unsuccessful**.

- ✓ There are two popular methods for searching the array elements: **linear search and binary search.**
- ✓ If the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

1. Linear Search

- ✓ Linear search, also called as **sequential search**, is a very simple method used for searching an array for a particular value.
- ✓ **It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.**
- ✓ Linear search is mostly used to search an **unordered list of elements** (array in which data elements are not sorted).
- ✓ For example, if an array A[] is declared and initialized as, int A[ ] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).
- ✓ Figure 14.1 shows the algorithm for linear search. In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

Example:

Write a program to search an element in an array using the linear search technique.

#include <stdio.h>

```
void main()
{
        int array[100], key, i, n;

        printf("Enter number of elements in array:\n");
        scanf("%d", &n);

        printf("Enter elements of array:\n");
        for (i = 0; i < n; i++)
                scanf("%d", &array [ i ] );

        printf("Enter a number to search\n");
        scanf("%d", &key);

        for (i = 0; i < n; i++)
        {
                if (array[i] == key)
                {
                        printf("%d is present at location %d.\n", key, i+1);
                        break;
                }
        }
        if (i == n)
                printf("%d isn't present in the array.\n", key);
}
```

```
Output:
Enter number of elements in
array:
5
Enter elements of array:
100
25
35
30
56
Enter a number to search:
30
30 is present at location 4
```

2. Binary Search

- ✓ Binary search is a **fast searching algorithm**.
- ✓ This search algorithm works on the principle of **divide and conquers**.
- ✓ Binary search is a searching algorithm that works efficiently with a **sorted list**.
- ✓ **Binary search begins by comparing the middle element of the array with the key value. If the key value matches the middle element, its position in the array is returned. If the key value is less than the middle element, the search continues in the lower half of the array. If the key value is greater than the middle element, the search continues in the upper half of the array.**
- ✓ Now, let us consider how this mechanism is applied to search for a value in a sorted array.
- ✓ In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (LOW + HIGH)/2. Initially, LOW = lower_bound and HIGH = upper_bound. The algorithm will terminate when KE = A[MID]. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array. However, if KEY is not equal to A[MID], then the values of LOW, HIGH, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If KEY < A[MID], then KEY will be present in the left segment of the array. So, the value of HIGH will be changed as HIGH = MID – 1.
(b) If KEY > A[MID], then KEY will be present in the right segment of the array. So, the value of BEG will be changed as LOW = MID + 1.

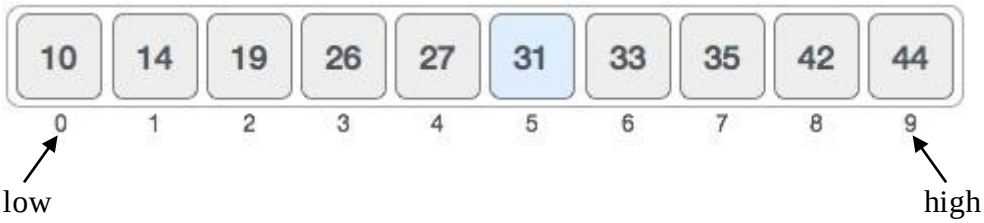
(c) Finally, if KEY is not present in the array, then eventually, HIGH will be less than LOW. When this happens, the algorithm will terminate and the search will be unsuccessful. Figure 14.2 shows the algorithm for binary search.

BINARY_SEARCH(A, lower_bound, upper_bound, KEY)
Step 1: [INITIALIZE] SET LOW = lower_bound
HIGH = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while LOW <= HIGH
Step 3: SET MID = (LOW + HIGH)/2
Step 4: IF KEY = A[MID]
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF KEY > A[MID]
SET LOW = MID + 1
ELSE
SET HIGH = MID - 1
[END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
Step 6: EXIT

How Binary Search Works?

- ✓ For a binary search to work, it is mandatory for the target array to be sorted. The following is our sorted array and let us assume that we need to search the location of key value **31** using binary search.

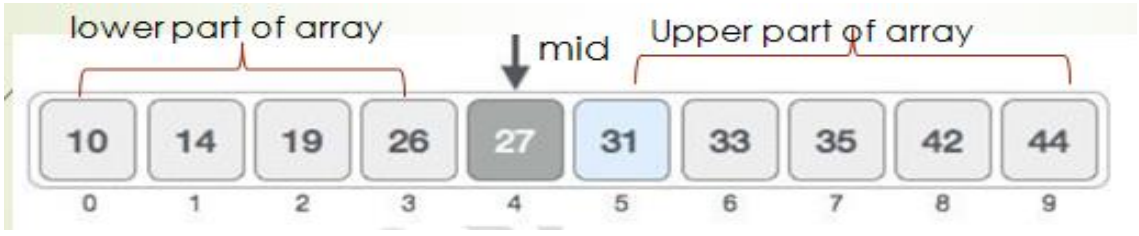
Here n=10, key=31



First, we shall determine middle position of the array by using this formula −

```
mid = (low + high) / 2
```

Here it is, mid=(0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

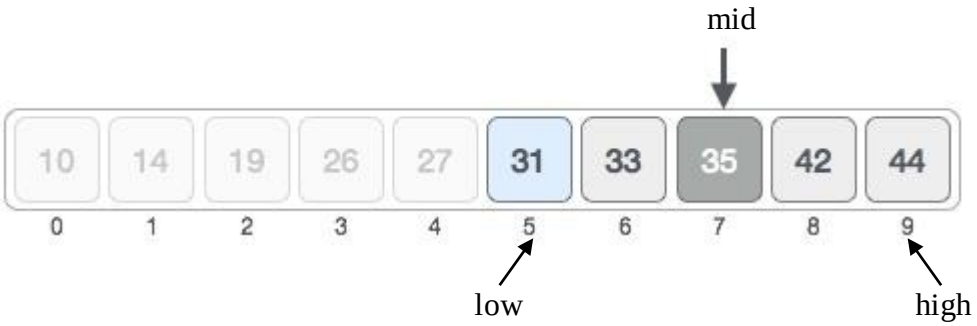


- ✓ Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. Since the key element is greater than the middle element, we should search the key element in the upper part of the array

We change our low to mid + 1 and find the new mid value again.

low = mid + 1=4+1=5
mid = (low + high) /2= (5+9)/2 =7

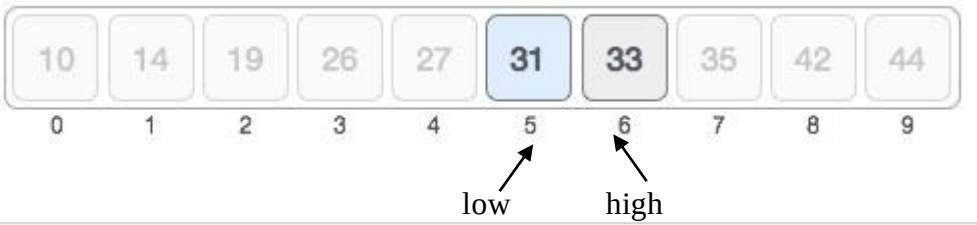
✓ Our new mid is 7 now. We compare the value stored at location 7 with our key value 31.



✓ The value stored at location 7 is not a match; rather it is more than what we are looking for. Since the key element is less than the middle element, we should search the key element in the lower part of the array.

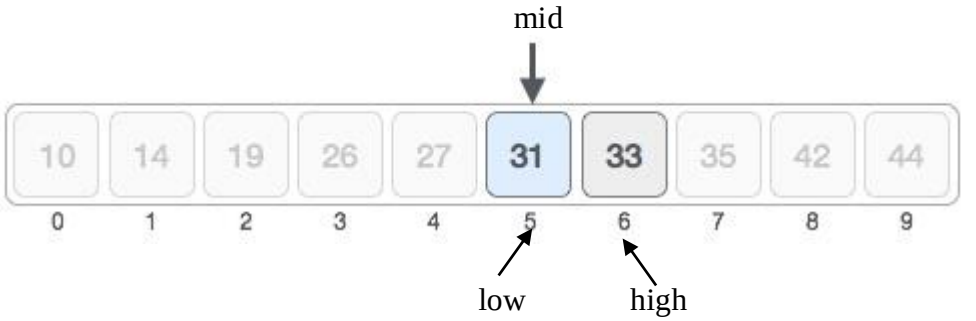
We change our high to mid - 1 and find the new mid value again.

high=mid-1 = mid -1=7-1=6



mid = (low + high) / 2 = (5+6)/2 =5

✓ Hence, we calculate the mid again. This time it is 5.



✓ We compare the value stored at location 5 with our key value. We find that it is a match.

✓ We conclude that the key value 31 is stored at position mid+1= 5+1=6.

- ✓ Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Example: C Program to search key elements in array using binary search algorithms.

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

void main()
{
        int  i, low, high, mid, n, key, a[100];
        printf("Enter number of elements in array:\n");
        scanf("%d",&n);
        printf("Enter integer numbers in ascending order:\n");
        for (i = 0; i< n; i++)
        scanf("%d",&a[i]);
        printf("Enter value to Search\n");
        scanf("%d", &key);
        low = 0;
        high = n - 1;
        while (low <= high)
        {
                mid = (low+high)/2;
                if (key == a[mid])
                {
                  printf("%d found at location %d.\n", key, mid+1);
                  exit(0);
                }
                if (key > a[mid] )
                        low = mid + 1;
                if (key < a[mid])
                        high = mid - 1;
        }
        printf(" %d is Not found! \n", key);

}
```

Output:
Enter number of elements in array:
5
Enter integer numbers in ascending order:
10
25
35
50
65
Enter a number to search:
65
65 is present at location 5

3.6 Passing Arrays to Functions

- ✓ Like variables of other data types, we can also pass an array to a function.
- ✓ In some situations, you may want to pass **individual elements of the array**; while in other situations, you may want to pass the **entire array** as shown in Fig. 3.20.

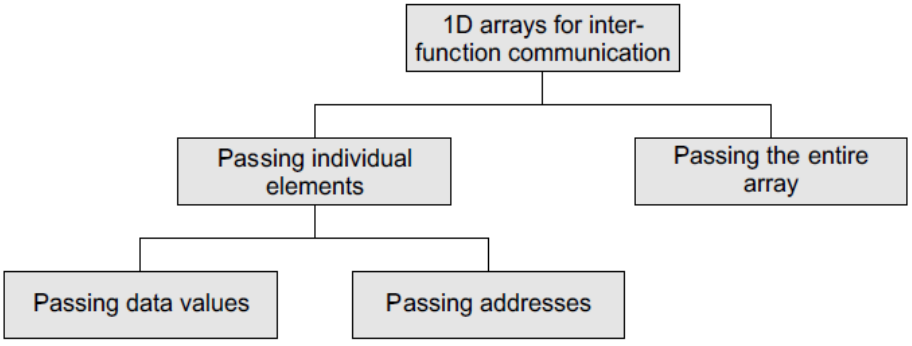


Figure 3.20 One dimensional arrays for inter-function communication

3.6.1 Passing Individual Elements

- ✓ The individual elements of an array can be passed to a function by passing either their **data values or addresses.**

1. Passing Data Values

- ✓ Individual elements can be passed in the same manner as we **pass variables of any other data type.**
- ✓ The condition is just that the **data type of the array element must match with the type of the function parameter.**
- ✓ Look at Fig. 3.21(a) which shows the code to pass an individual array element by passing the data value.

Calling function	Called function

`main()`
`{`
`  int arr[5] ={1, 2, 3, 4, 5};`
`  func(arr[3]);`
`}`
	`void func(int num)`
	`{`
	`  printf("%d", num);`
	`}`

Figure 3.21(a) Passing values of individual array elements to a function

- ✓ In the above example, only one element of the array is passed to the called function. This is done by using the index expression. Here, arr[3] evaluates to a single integer value.

2. Passing Addresses

- ✓ Like ordinary variables, we can pass the **address of an individual array element by preceding the indexed array element with the address operator**.
- ✓ Therefore, to pass the address of the fourth element of the array to the called function, we will write &arr[3].
- ✓ However, in the called function, the value of the array element must be accessed using the indirection (*) operator. Look at the code shown in Fig. 3.21(b).

Calling function	Called function

`main()`
`{`
`  int arr[5] ={1, 2, 3, 4, 5};`
`  func(&arr[3]);`
`}`
	`void func(int *num)`
	`{`
	`  printf("%d", *num);`
	`}`

Figure 3.21(b) Passing addresses of individual array elements to a function

3.6.2 Passing the Entire Array

- ✓ In C the array name refers to the first byte of the array in the memory.
- ✓ **The address of the remaining elements in the array can be calculated using the array name and the index value of the element.**
- ✓ Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array.
- ✓ Figure 3.22 illustrates the code which passes the entire array to the called function.

Calling function	Called function

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

Figure 3.22 Passing entire array to a function

3.7 Two-Dimensional Arrays

- ✓ **A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column.**
- ✓ The C compiler treats a two-dimensional array as an array of one-dimensional arrays.
- ✓ Figure 3.26 shows a two-dimensional array which can be viewed as an array of arrays.

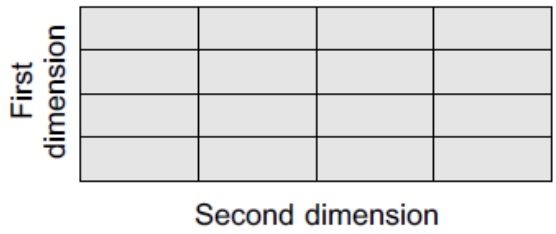


Figure 3.26 Two-dimensional array

3.7.1 Declaring Two-dimensional Arrays

- ✓ Any array must be declared before being used. The declaration statement tells the compiler the **name of the array, the data type of each element in the array, and the size of each dimension.**
- ✓ A two-dimensional array is declared as:

  data_type array_name[row_size][column_size];
- ✓ For example, if we want to store the marks obtained by three students in five different subjects, we can declare a two dimensional array as:

  int marks[3][5];
- ✓ In the above statement, a two-dimensional array called marks has been declared that has m(3) rows and n(5) columns. The first element of the array is denoted by marks[0][0], the second element as marks[0][1], and so on. Here, marks[0][0] stores the marks obtained by the first student in the first subject, marks[1][0] stores the marks obtained by the second student in the first subject.
- ✓ The pictorial form of a two-dimensional array is shown in Fig. 3.27.

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Figure 3.27 Two-dimensional array

- ✓ Hence, we see that a 2D array is treated as a collection of 1D arrays. Each row of a 2D array corresponds to a 1D array consisting of n elements, where n is the number of columns. To understand this, we can also see the representation of a two-dimensional array as shown in Fig. 3.28.

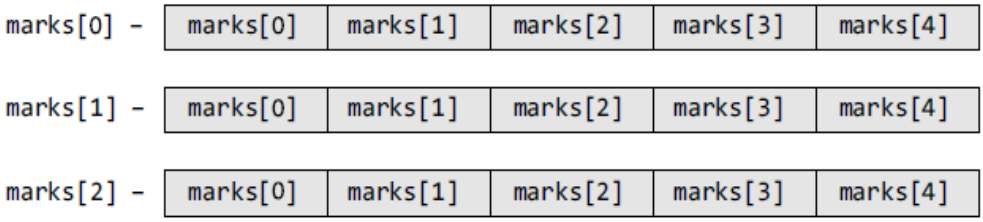


Figure 3.28 Representation of two-dimensional array `marks[3][5]`

- ✓ There are two ways of storing a two-dimensional array in the memory. The first way is the **row major order** and the second is the **column major order**.
- ✓ In a **row major order**, **the elements of the first row are stored before the elements of the second and third rows.** That is, the elements of the array are stored row by row where n elements of the first row will occupy the first n locations. This is illustrated in Fig. 3.29.

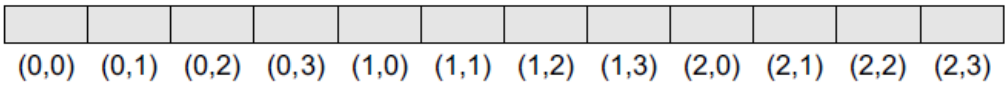


Figure 3.29 Elements of a 3 × 4 2D array in row major order

- ✓ However, when we store the elements in a **column major order, the elements of the first column are stored before the elements of the second and third column.** That is, the elements of the array are stored column by column where m elements of the first column will occupy the first m locations. This is illustrated in Fig. 3.30.

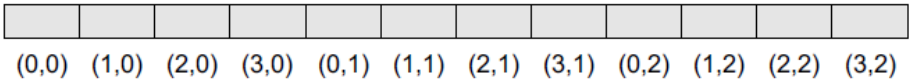


Figure 3.30 Elements of a 4 × 3 2D array in column major order

- ✓ If the array elements are stored in column major order,

Address(A[I][J]) = Base_Address + w{M (J – 1) + (I – 1)}

And if the array elements are stored in row major order,

Address(A[I][J]) = Base_Address + w{N (I – 1) + (J – 1)}

where w is the number of bytes required to store one element, N is the number of columns, M is the number of rows, and I and J are the subscripts of the array element.

Example 3.5 Consider a 20 × 5 two-dimensional array `marks` which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J]) = Base_Address + w{N (I - 1) + (J - 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 - 1) + (4 - 1)}
                      = 1000 + 2 {5(17) + 3}
                      = 1000 + 2 (88)
                      = 1000 + 176 = 1176
```

3.7.2 Initializing Two-Dimensional Arrays

- ✓ A two-dimensional array is initialized in the **same way as a one-dimensional array is initialized**. For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};
```

- ✓ Note that the initialization of a two-dimensional array is done row by row. The above statement can also be written as:

```
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

- ✓ The above two-dimensional array has two rows and three columns. First, the elements in the first row are initialized and then the elements of the second row are initialized. Therefore,

marks[0][0] = 90 marks[0][1] = 87 marks[0][2] = 78

marks[1][0] = 68 marks[1][1] = 62 marks[1][2] = 71

- ✓ In case of one-dimensional arrays, we have discussed that if the array is completely initialized, we may omit the size of the array. The same concept can be applied to a two-dimensional array, except that only the **size of the first dimension can be omitted**. Therefore, the declaration statement given below is valid.

```
int marks[][3]={{90,87,78},{68, 62, 71}};
```

- ✓ In order to initialize the **entire two-dimensional array to zeros**, simply specify the first value as zero. That is,

```
int marks[2][3] = {0};
```

- ✓ The **individual elements** of a two-dimensional array can be initialized using the **assignment operator** as shown here.

```
marks[1][2] = 79;
or
marks[1][2] = marks[1][1] + 10;
```

3.7.3 Accessing the Elements of Two-dimensional Arrays

- ✓ **The elements of a 2D array are stored in contiguous memory locations.**
- ✓ Since the two-dimensional array contains two subscripts, we will use **two for loops to scan the elements**.
- ✓ The first for loop will scan each row in the 2D array and the second for loop will scan individual columns for every row in the array.

Example:
Write a C program to read and print the elements of a 2D array.

```
#include <stdio.h>

void main()
{
        int arr[2][2], i, j, m, n;
        printf("Enter the size of the array:");
        scanf("%d%d",&m,&n);
        printf("Enter the elements of the array:\n);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("The elements of the array are:\n);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t", arr[i][j]);
                }
                printf("\n");
        }
}
```

3.8 Operations on Two-Dimensional Arrays

- ✓ Two-dimensional arrays can be used to **implement the mathematical concept of matrices**.
- ✓ In mathematics, a **matrix is a grid of numbers, arranged in rows and columns.**
- ✓ Thus, using two dimensional arrays, we can perform the following operations on an m×n matrix:

1. Transpose

- ✓ Transpose of an m × n matrix A is given as a n × m matrix B, where

$$B_{i,j} = A_{j,i}.$$

2. Sum

- ✓ Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

3. Difference

- ✓ Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing:

$$C_{i,j} = A_{i,j} - B_{i,j}$$

4. Product

- ✓ Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, m × n matrix A can be multiplied with a p × q matrix B if n=p. The dimension of the product matrix is m × q. The elements of two matrices can be multiplied by writing:

$$C_{i,j} = \Sigma\, A_{i,k}\, B_{k,j} \text{ for } k = 1 \text{ to } n$$

Example:

1. Write a C program to transpose a 3 × 3 matrix.

```
#include <stdio.h>

void main()
{
        int i, j, mat[3][3], transposed_mat[3][3];
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d", &mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
```

```
                printf("%d\t", mat[i][j]);
            }
            printf("\n");
        }
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
                transposed_mat[i][j] = mat[j][i];
        }


        printf("\n The elements of the transposed matrix are ");
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
            {
                printf("%d\t",transposed_ mat[i][j]);
            }
            printf("\n");
        }
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
The elements of the transposed matrix are
1 4 7
2 5 8
3 6 9
```

2. Write a program to input two m × n matrices and then calculate the sum of their corresponding elements and store it in a third m × n matrix.

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

void main()
{
    int i, j, m, n,  p, q, a[5][5], b[5][5], c[5][5];
    printf("\n Enter the number of rows and columns in the first matrix : ");
    scanf("%d%d",&m,&n);

    printf("\n Enter the number of rows and columns in the second matrix : ");
    scanf("%d%d",&p,&q);

    if(m != p || n != q)
    {
        printf("\n Number of rows and columns of both matrices must be equal");
        exit(0);
```

```
        }

        printf("\n Enter the elements of the first matrix ");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }

        printf("\n Enter the elements of the second matrix ");
        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                {
                        scanf("%d",&b[i][j]);
                }
        }

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        c[i][j] = a[i][j] + b[i][j];
        }

        printf("\n The elements of the resultant matrix are ");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t ", c[i][j]);
                }
                printf("\n");
        }
}
```

3.9 Passing two-dimensional arrays to functions

- ✓ There are three ways of passing a two-dimensional array to a function.
- ✓ First, we can **pass individual elements of the array**. This is exactly the same as passing an element of a one-dimensional array.
- ✓ Second, we can pass a **single row of the two-dimensional array.** This is equivalent to passing the entire one-dimensional array to a function.
- ✓ Third, we can pass the **entire two-dimensional array to the function.**
- ✓ Figure 3.31 shows the three ways of using two-dimensional arrays for inter-function communication.

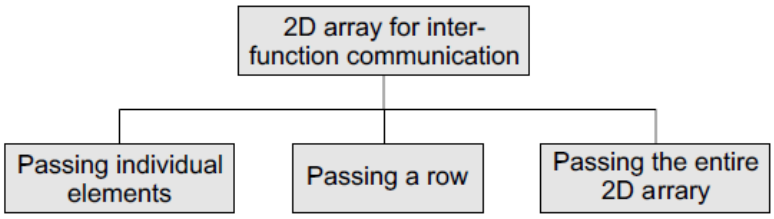


Figure 3.31 2D arrays for inter-function communication

1. Passing individual elements

✓ The individual elements of an array can be passed to a function by passing either their **data values or addresses.**

Passing Data Values

Calling function

```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(arr[1][1]);
}
```

Called function

```
void func(int num)
{
        printf("%d", num);
}
```

Passing Addresses

Calling function

```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(&arr[1][1]);
}
```

Called function

```
void func(int *num)
{
        printf("%d", *num);
}
```

2. Passing a Row

✓ A row of a two-dimensional array can be passed by **indexing the array name with the row number.**

✓ Look at Fig. 3.32 which illustrates how a single row of a two-dimensional array can be passed to the called function.

Calling function

```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(arr[1]);
}
```

Called function

```
void func(int arr[])
{
        int i;
        for(i=0;i<5;i++)
            printf("%d", arr[i]);
}
```

3. Passing the Entire 2D Array

✓ To pass a two-dimensional array to a function, we use the **array name as the actual parameter**. However, the parameter in the called function must indicate that the array has two dimensions.

Calling function

```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(arr);
}
```

Called function

```
void func(int arr[5][5])
{
        for(i=0;i<5;i++)
                for(j=0;j<5;j++)
                    printf("%d", arr[i][j]);
}
```

3.10 Multi-Dimensional Arrays

- ✓ A multi-dimensional array in simple terms is **an array of arrays.**
- ✓ As we have **one index in a one-dimensional array**, **two indices in a two-dimensional array**, in the same way, we have **n indices in an n-dimensional array or multi-dimensional array.**
- ✓ Conversely, an **n–dimensional array is specified using n indices**.
- ✓ An n-dimensional $m_1 \times m_2 \times m_3 \times ... \times m_n$ array is a collection of $m_1 \times m_2 \times m_3 \times ... \times m_n$ elements. In a multi-dimensional array, a particular element is specified by using n subscripts as $A[I_1][I_2][I_3]...[I_n]$,
- ✓ Figure 3.33 shows a three-dimensional array. The array has three pages, four rows, and two columns.

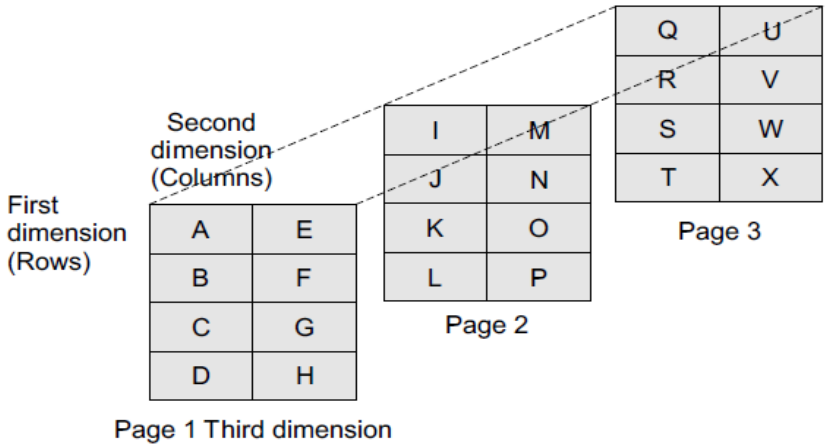


Figure 3.33 Three-dimensional array

Example:

1. Write a program to read and display a 2 × 2 × 2 array.

```
#include <stdio.h>

void main()
{
        int array[2][2][2], i, j, k;

        printf("\n Enter the elements of the matrix");
        for(i=0;i<2;i++)
        {
                for(j=0;j<2;j++)
                {
                        for(k=0;k<2;k++)
                        {
                                scanf("%d", &array[i][j][k]);
                        }
                }
        }
```

```
        printf("\n The matrix is : ");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                {
                        printf("\n");
                        for(k=0;k<2;k++)
                                printf("\t array[%d][%d][%d] = %d", i, j, k, array[i][j][k]);
                }
        }
}
```

Output

Enter the elements of the matrix
1 2 3 4 5 6 7 8
The matrix is
arr[0][0][0] = 1 arr[0][0][1] = 2
arr[0][1][0] = 3 arr[0][1][1] = 4
arr[1][0][0] = 5 arr[1][0][1] = 6
arr[1][1][0] = 7 arr[1][1][1] = 8

3.11 Applications of Arrays

Arrays are frequently used in C, as they have a number of useful applications. These applications are

- ✓ Arrays are widely used to **implement mathematical vectors, matrices, and other kinds of rectangular tables.**
- ✓ **Many databases** include one-dimensional arrays whose **elements are records.**
- ✓ Arrays are also used to **implement other data structures such as strings, stacks, queues, heaps, and hash tables.**
- ✓ Arrays can be used for **sorting elements in ascending or descending order.**

MODULE 3
Functions

3.12 Introduction

- ✓ C enables its programmers to **break up a program into segments** commonly known as **functions**, each of which can be written **more or less independently of the others**.
- ✓ **Every function in the program is supposed to perform a well-defined task.**
- ✓ Therefore, the program code of one function is completely insulated from the other functions.

Definition

- ✓ **"The set of instructions that performs some specific, well-defined task is called as a Function."**

Or

- ✓ **"Function is a small program or program segment that carryout some specific well-defined tasks".**
- ✓ Fig. 1.9 explains how the main() function calls another function to perform a well-defined task.
- ✓ In the figure, we can see that main() calls a function named func1(). Therefore, main() is known as the **calling function** and func1() is known as the **called function**.
- ✓ The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function.
- ✓ After the called function is executed, the control is returned to the calling program.

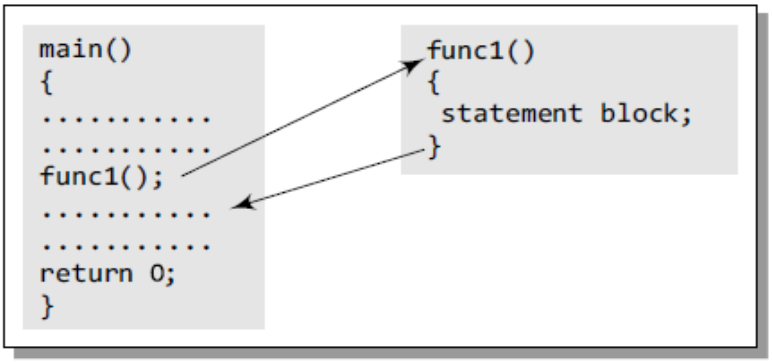


Figure 1.9 main() calls func1()

3.12.1 Why are Functions Needed?

- ✓ **Dividing the program** into separate **well-defined functions** facilitates each function to be **written and tested separately.** This **simplifies the process of getting the total program to work.**
- ✓ **Understanding, coding, and testing** multiple separate functions is easier than doing the same for one big function.
- ✓ If a **big program** has to be developed without using any function other than main(), then there will be **countless lines** in the main() function and **maintaining that program** will be a **difficult task.**
- ✓ All the **libraries in C contain a set of functions**, which have been **pre-written and pre-tested**, so the **programmers can use them without worrying about their code details.** This **speeds up program development**, by allowing the programmer to concentrate only on the code that he has to write.

- ✓ Like C libraries, **programmers can also write their own functions and use them** from different points in the main program or any other program that needs its functionalities.
- ✓ When a big program is broken into comparatively smaller functions, then different programmers working on that project can **divide the workload by writing different functions.**

3.13 Using Functions

While using functions, we will be using the following terminologies:

- ✓ A function **f** that **uses another function g** is known as the **calling function**, and **g** is known as the **called function**.
- ✓ The **inputs that a function takes** are known as **arguments**.
- ✓ When a **called function returns some result back** to the calling function, it is said to **return** that **result.**
- ✓ The **calling function may or may not pass parameters** to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- ✓ **Function declaration** is a declaration statement that identifies **a function's name, a list of arguments that it accepts, and the type of data it returns.**
- ✓ **Function definition** consists of a **function header** that identifies the function, followed by the **body of the function** containing the executable code for that function.

3.14 Types of Functions

i. Library Functions/Pre-Defined/ Built-in Functions

- ✓ C Library of C Compiler has a collection of various functions which perform some standard and pre-defined tasks.
- ✓ These functions written by designers of C Compilers are called as Library functions/Pre-Defined/Built-in functions.

Ex: sqrt(n)- computes square root of n.
pow(x,y)- computes x^y.
printf()- used to print the data on the screen.
scanf()- used to read the data from the keyboard.
abs(x)- computes absolute value of x.

ii. User-Defined/ Programmer Defined Functions

- ✓ The functions written by the programmer/user to do the specific tasks is called User-Defined/ Programmer Defined Functions.
- ✓ main( ) is the user defined function.

3.15 Elements of User-Defined Functions

- ✓ The three elements of user-defined functions are shown below:
  i. Function Prototype/Declaration
  ii. Function Definition
  iii. Function Call

3.15.1 Function Declaration/Function Prototype

- ✓ Before using a function, the compiler must know the **number of parameters and the type of parameters** that the function expects to receive and the **data type of value** that it will return to the calling program.
- ✓ Placing the function declaration statement **prior to its use** enables the compiler to make a **check on the arguments used** while calling that function.
- ✓ The general format for declaring a function that accepts arguments and returns a value as result can be given as:

return_data_type function_name(data_type variable1, data_type variable2,..);

Here, **function_name** is a **valid name for the function**. A function should have a meaningful name that must specify the task that the function will perform.

return_data_type specifies the **data type of the value that will be returned** to the calling function as a result of the processing performed by the called function.

(data_type variable1, data_type variable2, ...) is a **list of variables** of specified data types. These variables are passed from the calling function to the called function. They are also known as **arguments or parameters** that the called function accepts to perform its task.

Ex: int add(int a,int b);

- ✓ Things to remember about function declaration:
  - After the declaration of every function, there should be a **semicolon.** If the semicolon is missing, the compiler will generate an error message.
  - The function declaration is **global.**
  - **Use of argument name** in the function declaration is **optional.**
    int func(int, char, floa t);
    or
    int func(int num, char ch, float fnum);
  - A function **cannot be declared** within the body of another function.
  - A function having **void as its return type cannot return any value.**
  - A function having void as its parameter list **cannot accept any value.** So the function declared as
    void print();
    does not accept any input/arguments from the calling function .
  - If the function declaration does not specify any return type, then by default, the function **returns an integer value.** Therefore, when a function is declared as
    sum(int a, int b);
    Then the function sum accepts two integer values from the calling function and in sum returns an integer value to the caller.
  - Some compilers make it compulsory to declare the function before its usage while other compilers make it optional.

3.15.2 Function Definition

- ✓ When a **function is defined**, **space is allocated** for that function in the memory.
- ✓ A function definition comprises of two parts:
  - **Function header**
  - **Function body**

✓ The syntax of a function definition can be given as:

return_data_type function_name(data_type variable1, data_type variable2,..)

```
{
        .............
        statements
        .............
        return(variable);
}
```

✓ The **number of arguments and the order of arguments** in the function header must be the **same as that given in the function declaration** statement.

✓ While **return_data_type function_name(data_type variable1, data_type variable2,...)** is known as the **function header**, the rest of the portion comprising of **program statements within the curly brackets { } is the function body** which contains the code to perform the specific task.

✓ Note that the function header is same as the function declaration. The only difference between the two is that a **function header is not followed by a semi-colon.**

Ex:
```
int add(int a,int b)
 {
        int sum;
        sum=a+b;
        return sum;
 }
```

3.15.3 Function Call

✓ The function call statement **invokes the function.**

✓ When a function is invoked, the **compiler jumps to the called function** to **execute the statements** that are a part of that function.

✓ Once the called function is executed, the program control passes back to the calling function. A function call statement has the following syntax:

function_name(variable1, variable2, ...);

✓ The following points are to be noted while calling a function:

- **Function name and the number and the type of arguments** in the **function call** must be **same** as that given in the **function declaration and the function header** of the function definition.
- **Names** (and not the types) of variables in **function declaration, function call, and header of function definition may vary.**
- **Arguments** may be passed in the **form of expressions** to the called function. In such a case, **arguments are first evaluated** and converted to the type of formal parameter and then the body of the function gets executed.
- If the **return type** of the function is **not void**, then the value returned by the called function may be assigned to some variable as given below.

variable_name = function_name(variable1, variable2, ...);

Ex: add(a,b);

3.16 return STATEMENT

- ✓ **The return statement terminates the execution of the called function and returns control to the calling function.**
- ✓ When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- ✓ A return statement **may or may not return a value to the calling function.**
- ✓ The syntax of return state can be given as

return <expression>;

Here expression is placed in between angular brackets because specifying an expression is optional.

- ✓ A function that has **void return type cannot return any value** to the calling function.

3.17 Passing Parameters to Functions

- ✓ There are **two ways** in which arguments or parameters can be passed to the called function.

Call by value: The **values of the variables** are passed by the calling function to the called function.

Call by reference The **addresses of the variables** are passed by the calling function to the called function.

1. Call by Value

- ✓ In this method, the **called function creates new variables to store the value of the arguments passed to it.** Therefore, the called function uses a **copy of the actual arguments** to perform its intended task.
- ✓ **If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function, no change will be made to the value of the variables. This is because all the changes are made to the copy of the variables and not to the actual variables.**

Example: Write a C program to add two numbers using call by value.

```
#include<stdio.h>
int add (int a,int b)
{
        int  sum;
        sum = a + b;
        return sum;
}

void main()
{
        int a,b, res;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        res = add(a,b);
        printf("result =%d\n", res);
}
```

Output:
Enter the values of a and b: 4 5
result =9

- ✓ Following are the points to remember while passing arguments to a function using the call-by value method:
  - **When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.**
  - **The values of the arguments passed by the calling function are copied into the newly created variables.**
  - **Values of the variables in the calling functions remain unaffected when the arguments are passed using the call-by-value technique.**

Pros and cons

- ✓ The biggest advantage of using the call-by-value technique is that **arguments** can be passed as **variables, literals, or expressions**, while its main drawback is that **copying data consumes additional storage space.**
- ✓ In addition, it can take a **lot of time to copy**, thereby resulting in **performance penalty**, especially if the function is called many times.

2. Call by Reference

- ✓ **When the calling function passes arguments to the called function using the call-by-value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement.** A better option is to pass arguments using the call-by-reference technique.
- ✓ In this method, we declare the **function parameters as references** rather than normal variables.
- ✓ **When this is done, any changes made by the function to the arguments it received are also visible in the calling function.**
- ✓ To indicate that an argument is passed using call by reference, an **asterisk (*)** is placed after the type in the parameter list. Hence, in the call-by-reference method, a function **receives an implicit reference to the argument**, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well.

Example:

1. Write a C program to add two numbers using call by reference.

```
#include<stdio.h>

int add (int *a,int *b)
{
        int  sum;
        sum = *a + *b;
        return sum;
}

void main()
{
        int a,b, res;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        res = add(&a,&b);
        printf("result =%d\n", res);
}
```

Output:

Enter the values of a and b: 4 5
result =9

2. Write a C program to swap two numbers using call by reference.

```
#include<stdio.h>
void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
void main()
{
        int a,b;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        printf("Before swapping: a=%d\tb=%d", a, b);
        swap(&a,&b);
        printf("After swapping: a=%d\tb=%d", a, b)
}
```

```
Enter the values of a and b: 10 20
Before swapping: a=10        b=20
After swapping: a=20         b=10
```

Output:

Enter the values of a and b: 10 20
Before swapping: a=10 b=20
After swapping: a=20 b=10

Advantages

- ✓ Since arguments are not copied into the new variables, it provides **greater time and space efficiency.**
- ✓ The **function can change the value** of the argument and the change is reflected in the calling function.
- ✓ A function can **return only one value**. In case we need **to return multiple values**, we can pass those arguments by reference, so that the modified values are visible in the calling function.

Disadvantages

- ✓ However, the drawback of using this technique is that if **inadvertent changes** are caused to variables in called function then these **changes would be reflected in calling function** as **original values would have been overwritten.**

3.18 Scope of Variables

- ✓ In C, all **constants and variables** have a **defined scope**.
- ✓ By scope we mean the **accessibility and visibility of the variables at different points in the program.**
- ✓ A variable or a constant in C has **four types of scope: block, function, program, and file.**

3.18.1 Block Scope

- ✓ We have studied that a **statement block is a group of statements enclosed within opening and closing curly brackets { }.**
- ✓ **If a variable is declared within a statement block then as soon as the control exits that block, the variable will cease to exist.**
- ✓ Such a variable also known as a **local variable is said to have a block scope.**
- ✓ So far we had been using local variables.
- ✓ For example, if we declare an integer x inside a function, then that variable is unknown to the rest of the program (i.e., outside that function).
- ✓ Variables declared with **same names** as those in outer block **mask the outer block variables** while executing the inner block.
- ✓ In nested blocks, **variables declared outside the inner blocks are accessible to the nested blocks**, provided these variables are not re-declared within the inner block.

3.18.2 Function Scope

- ✓ Function scope indicates that a **variable is active and visible from the beginning to the end of a function.**
- ✓ In C, only the **goto label** has function scope.
- ✓ In other words function scope is applicable only with goto label names. This means that the programmer cannot have the same label names inside a function.

Ex:

```
void main()
{
        …
        …
        loop: /*A goto label has function scope */
        …
        …
        goto loop /* the goto statement */
        …
        …
}
```

- ✓ In this example, the **label loop is visible from the beginning to the end of the main () function.** Therefore, there **should not be more than one label having the same name** within the main() function.

3.18.3 Program Scope

- ✓ Till now we have studied that variables declared within a function are **local variables**. These local variables (also known as internal variables) are **automatically created** when they are **declared in the function** and are **usable only within that function**. The local variables are **unknown to other functions** in the program. Such variables **cease to exist after the function** in which they are declared is exited and are re-created each time the function is called.
- ✓ However, if you want a **function to access some variables which are not passed to it as arguments**, then declare those variables outside any function blocks. Such variables are commonly known as **global variables** and can be accessed from any point in the program.

Lifetime: Global variables are created at the **beginning of program execution** and remain in existence throughout the **period of execution of the program**. These variables are **known to all the functions** in the program and **are accessible** to them for usage. Global variables are not limited to a particular function so they exist even when a function calls another function. These variables retain their value so that they can be used from every function in the program.

Place of Declaration: The Global variables are **declared outside all the functions including main().** It is always recommended to declare them on **top of the program code.**

Name conflict: If we have a variable declared in a function that has **same name as that of the global variable**, then the function will use the **local variable declared within** it and **ignore** the global variable. Consider the following program,

```
#include <stdio.h >
int x = 10;
void print();

void main()
{
        printf("\n The value of x in the main() = %d", x);
        int x = 2;
        printf("\n The value of local variable x in the main() = %d", x);
        print();
}

void print()
{
        printf("\n The value of x in the print() = %d", x);
}
```

Output:

```
The value of x in the main() = 10
The value of local variable x in the main()= 2
The value of x in the print () = 10
```

3.18.4 File Scope

- ✓ When a global variable is **accessible until the end of the file**, the Variable is said to have **file scope**. To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type:

  static int x;

- ✓ A global static variable can be used **anywhere from the file** in which it is declared but it is **not accessible by any other file.**

3.19 Storage Classes

- ✓ **Storage class defines the scope (visibility) and lifetime of variables and/or functions declared within a C program.**
- ✓ In addition to this, the storage class gives the following information about the variable or the function.
  - The storage class of a function or a variable determines the **part of memory where storage space will be allocated for that variable or function** (whether the variable function will be stored in a **register or in RAM**).

- It specifies **how long the storage allocation will continue to exist** for that function or variable.
- It specifies the **scope of the variable or function**.
- It specifies whether the variable or function has **internal, external, or no linkage.**
- It specifies whether the variable will be **automatically initialized to zero or to any indeterminate value.**

✓ C supports four storage classes: **automatic, register, external, and static.**

✓ The general syntax for specifying the storage class of a variable can be given as:

<storage_class_specifier> <data type > <variable name>

3.19.1 auto Storage Class

✓ The auto storage class specifier is used to explicitly declare a variable with **automatic storage**.

✓ It is the **default storage class** for variables declared inside a block.

✓ For example, if we write

auto int x;

then x is an integer that has automatic storage. It is deleted when the block in which x is declared is exited.

✓ The auto storage doss can be used to declare **variables in a block or the names of function parameters.**

✓ Important things to remember about the variables declared with **auto** storage class are as follows :

- **All the variables declared within a function** belong to automatic storage class by default.
- They should be declared at the **start of the program block**, right after the opening curly brackets {.
- **Memory** for the variable is **automatically allocated upon entry to a block** and freed automatically upon exit from that block.
- The scope of the variable is **local to the block** in which it is declared.
- Every time the block is entered, the **variable is initialized with the values declared.**
- The auto variables are stored in the **primary memory of the computer.**
- If auto variables are **not initialized** at the time of declaration, then they contain **some garbage value.**

3.19.2 register Storage Class

✓ When a variable is declared using register as its storage class, it is stored in **a CPU register** instead of RAM.

✓ Since the variable is stored in a register, the **maximum size of the variable is equal to the register size.**

✓ A register variable is declared in the following manner:

register int x;

✓ Register variables are used when **quick access to the variable** is needed.

✓ Each time the **block is entered**, the register variables defined in that **block are accessible** and the moment that **block is exited**, the variables become **no longer accessible for use**.

3.19.3 extern Storage Class

✓ The **extern** storage class is used to give a **reference of a global variable** that is visible to all the program files.

✓ Such global variables are declared like any other variables in one of the program files. .

✓ To declare a variable x as extern write,

extern int x;

- ✓ External variables may be declared outside any function source code file as any other variable is declared.
- ✓ Usually external variables are declared and defined in the **beginning of a source file.**
- ✓ **Memory is allocated for the external variables when the program begins execution and remains allocated until the program terminates.**
- ✓ In case if the external variable is not initialized, then it will be **initialized to zero by default.**
- ✓ External variables have **global scope,** i.e. these variables are visible and accessible from all the functions in the program.

3.19.4 static Storage Class

- ✓ static is the **default storage class for all global variables.**
- ✓ Static variables have a **lifetime over the entire program**. i.e., memory for the static variables is allocated when the program begins running and is freed when the program terminates.
- ✓ To declare an integer x as static, write

```
static int x = 10;
```

Here x is a local static variable.

- ✓ Static local variables when defined within a function are **initialized at the runtime.**
- ✓ The static variables are initialized just once, when defined within a function it is not re-initialized when the function is called again and again.
- ✓ When a **static variable is not explici1ly initialized** by the programmer, then it is automa1ically **initialized to zero** when memory is allocated for it

Comparison of Storage Classes

Table 11.2 Comparison of storage classes

FEATURE	STORAGE CLASS			
	auto	extern	register	static
Accessibility	Accessible within the function or block in which it is declared.	Accessible within all program files that are a part of the program.	Accessible within the function or block in which it is declared.	Local: Accessible within the function or block in which it is declared. Global: Accessible within the program in which it is declared.
Storage	Main memory	Main memory	CPU register	Main memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Exists throughout the execution of the program.	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Local: Retains value between function calls or block entries.Global: Preserves value in program files.
Default value	Garbage	Zero	Garbage	Zero

3.20 Recursion

- ✓ **"The process in which a function calls itself again and again is called as Recursion".**
- ✓ **A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.**

✓ Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are:

- **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Ex: To calculate n!, we multiply the number with factorial of the number that is 1 less than that number. In other words, $n! = n \times (n-1)!$

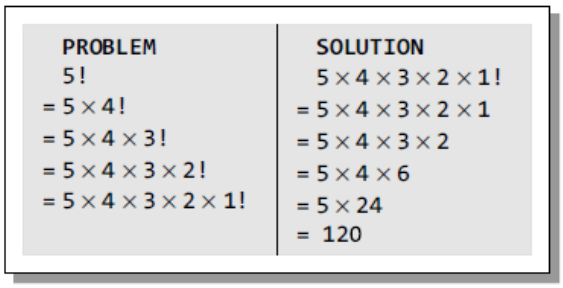


Figure 7.27 Recursive factorial function

✓ Every recursive function must have a base case and a recursive case. For the factorial function,

- **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
- **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as:

factorial(n) = n × factorial (n–1)

Example:

1. Write a C program to calculate factorial of a given number.

```
#include<stdio.h>
int factorial(int n)
{
        if(n==1)
                return 1;
        else
                 return (n*fact(n-1));
}

void main()
{
        int n,fact;
        printf("Enter a number:");
        scanf("%d",&n);
        fact=factorial(n);
        printf("\nFactorial of given number=%d",fact);
}
```

Output:
Enter a number: 5
Factorial of given number =120

Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using the **Euclid's algorithm** that states:

$$GCD\ (a,\ b) = \begin{cases} b, & \text{if } b \text{ divides } a \\ GCD\ (b,\ a \bmod b), & \text{otherwise} \end{cases}$$

Working

Assume a = 62 and b = 8

```
GCD(62, 8)
   rem = 62 % 8 = 6
   GCD(8, 6)
          rem = 8 % 6 = 2
          GCD(6, 2)
                 rem = 6 % 2 = 0
          Return 2
   Return 2
Return 2
```

2. Write a C program to calculate the GCD of two numbers using recursive functions.

```
#include <stdio.h>
int GCD(int x, int y)
{
        int rem;
        rem = x%y;
        if(rem==0)
                return y;
        else
                return (GCD(y, rem));
}

void main()
{
        int x, y, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &x, &y);
        res = GCD(x, y);
        printf("\n GCD = %d", x, y, res);
}
```

Output:
Enter the two numbers: 8 12
GCD = 4

3. Write a C program to find the Fibonacci series using recursive function.

```
#include<stdio.h>
int fibonacci (int n)
{
    if( n = = 0)
       return 0;
    else if (n = = 1)
       return 1;
    else
       return ( fibonacci (n-1) + fibonacci (n-2));
}
```

```
void main()
{
    int n, i, res;
    printf("Enter a value for n:");
    scanf("%d",&n);
    printf(" The Fibonacci series is: \n");
    for ( i =0; i < n ; i ++)
    {
        res = fibonacci(i);
        printf("%d\n", res);
    }
}
```

Output:

```
Enter a value for n:
5
The Fibonacci series is:
0
1
1
2
3
```

3.21 Function Parameters

- ✓ **"The list of variables defined in the function header within the parenthesis are called Function parameters".**
- ✓ There are 2 types of parameters in 'C' functions.
  - i. Actual parameters
  - ii. Formal parameters

i. Actual or Real Parameters

- ✓ **The variables that are used when a function is invoked are called actual parameters.**
- ✓ Actual parameters are used in the Calling function when a function is invoked.
- ✓ Actual parameters send values or addresses to the formal parameters. Formal parameters receive them and use the same values.
- ✓ Actual parameters can be constants, variables or expressions.

Ex: res=add(m,n);

ii. Formal or Dummy Parameters

- ✓ **The variables defined in the function header or function definition are called formal parameters.**
- ✓ All the variables should be separately declared and each declaration must be separated by commas.
- ✓ The formal parameters receive values form the actual parameters.
- ✓ If the formal parameters receive the address from the actual parameters, then they should be declared as pointers.
- ✓ The formal parameters should be only variables. Expressions and constants are not allowed.

Ex: int add(int a,int b);

Example Program: C Program to define actual and formal parameters.

```
#include<stdio.h>

int add(int a,int b)                    // Formal Parameters a, b
{
        int sum;
        sum=a+b;
        return sum;
}
```

```
void main()
{
        int m,n,res;
        printf("Enter the values for m,n\n");
        scanf("%d%d",&m,&n);
        res=add(m,n);                          // Actual parameters m,n
        printf("Sum=%d\n",res);
}
```

MODULE 4
Strings

4.1 Introduction

- ✓ **"A string is a sequence of characters enclosed within double quotes". or**
- ✓ **"String is an array of characters and terminated by NULL character which is denoted by '\0'.**
- ✓ In C, a string is a null-terminated character array.
- ✓ This means that after the last character, a null character ('\0') is stored to signify the end of the character array.

For example, if we write char str[] = "HELLO";

we are declaring an array that has five characters, namely, H, E, L, L, and O. Apart from these characters, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes HELLO'\0'. To store a string of length 5, we need 5 + 1 locations (1 extra for the null character). The name of the character array (or the string) is a pointer to the beginning of the string.

- ✓ Figure 4.1 shows the difference between character storage and string storage.

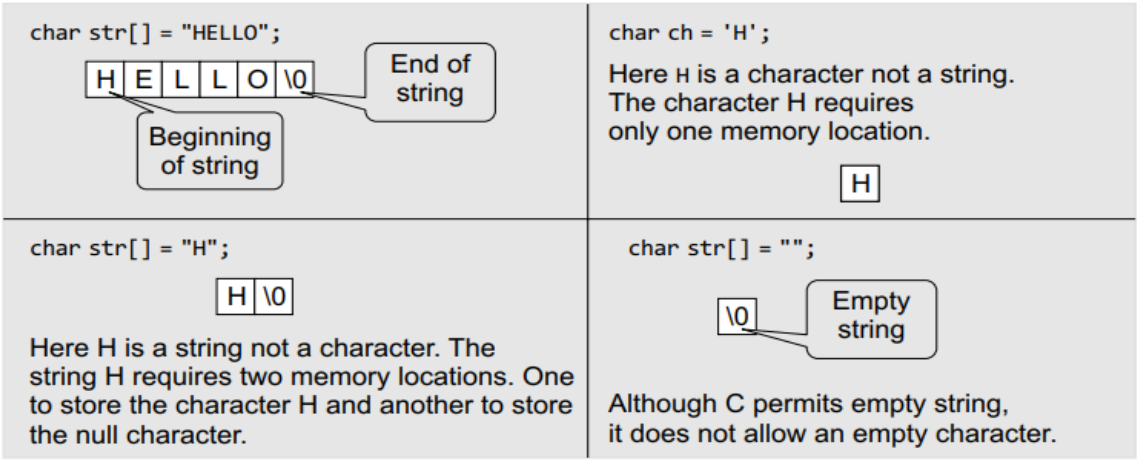


Figure 4.1 Difference between character storage and string storage

- ✓ If we had declared str as

  char str[5] = "HELLO";

then the null character will not be appended automatically to the character array. This is because str can hold only 5 characters and the characters in HELLO have already filled the space allocated to it.

- ✓ Like we use **subscripts** (also known as **index**) to **access the elements of an array**, we can also use **subscripts to access the elements of a string**. The subscript **starts with a zero (0).** All the characters of a string are stored in successive memory locations. Figure 4.2 shows how str[] is stored in the memory.

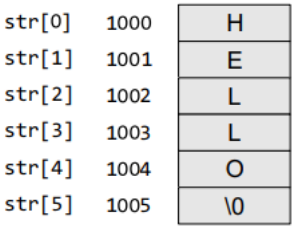


Figure 4.2 Memory representation of a character array

- ✓ ASCII code of a character is stored in the memory and not the character itself. So, at address 1000, 72 will be stored as the ASCII code for H is 72. The statement char str[] = "HELLO";

  Syntax:

  the general form of declaring a string is

  char str[size];
- ✓ The other way to initialize a string is to initialize it as an **array of characters.** For example,

  char str[] = {'H', 'E', 'L', 'L', 'O', '\0'};

  Here, the compiler will automatically calculate the size based on the number of characters.
- ✓ We can also declare a string with **size much larger than the number of elements** that are initialized. For example, consider the statement below.

  char str [10] = "HELLO";

  In such cases, the compiler creates an array of size 10; stores "HELLO" in it and finally terminates the string with a null character. Rest of the elements in the array are automatically initialized to NULL
- ✓ Now consider the following statements:

  char str[3];

  str = "HELLO";

  The above initialization statement is **illegal in C and would generate a compile-time error**.

4.1.1 Reading Strings

- ✓ If we declare a string by writing char str[100]; Then str can be read by the user in three ways:
  1. using **scanf function**,
  2. using **gets() function**, and
  3. using **getchar(),getch()or getche()** function repeatedly.

using scanf()

- ✓ Strings can be read using scanf() by writing **scanf("%s", str);**
- ✓ Unlike int, float, and char values, %s format does not require the ampersand before the variable str.
- ✓ The main **pitfall** of using this function is that the **function terminates as soon as it finds a blank space. Therefore we cannot read the complete sentence using scanf()** function.

using gets()

- ✓ The string can be read by writing **gets(str);**
- ✓ gets() is a simple function that **overcomes the drawbacks of the scanf() function.**
- ✓ **gets() function is used to read a sequence of characters (string) with spaces in between.**
- ✓ The 'gets()' function allows us to read an 'entire line' of input including whitespace characters.
- ✓ The gets() function takes the starting address of the string which will hold the input.
- ✓ The string inputted using gets() is **automatically terminated with a null character.**

using getchar()

- ✓ Strings can also be read by calling the **getchar() function repeatedly to read a sequence of single characters** (unless a terminating character is entered) and simultaneously storing it in a character array as shown below:

```
i=0;
ch = getchar;// Get a character
while(ch != '*')
{
        str[i] = ch;// Store the read character in str
        i++;
        ch = getchar();// Get another character
}
str[i] = '\0';// Terminate str with null character
```

4.1.2 Writing Strings

- ✓ Strings can be displayed on the screen using the following three ways:
  1. using **printf() function**
  2. using **puts() function**, and
  3. using **putchar() function** repeatedly.

using printf()

- ✓ Strings can be displayed using printf() by writing **printf("%s", str);**
- ✓ We use the **format specifier %s** to output a string. Observe carefully that there is **no '&' character** used with the string variable.
- ✓ We may also use **width and precision specifications** along with %s.
- ✓ The precision specifies the maximum number of characters to be displayed, after which the string is truncated. For example, printf ("%5.3s", str); The above statement would print only the first three characters in a total field of five characters. Also these characters would be right justified in the allocated width.
- ✓ To make the string left justified, we must use a minus sign. For example, printf ("%–5.3s", str);

using puts()

- ✓ A string can be displayed by writing **puts(str);**
- ✓ puts() is a simple function that overcomes the drawbacks of the printf() function.
- ✓ The puts() function **writes a line of output on the screen.** It terminates the line with a newline character ('\n').

using putchar()

- ✓ Strings can also be written by calling the **putchar() function repeatedly to print a sequence of single characters.**

```
i=0;
while(str[i] != '\0')
{
        putchar(str[i]);// Print the character on the screen
        i++;
}
```

4.2 String Taxonomy

- ✓ In C, we can store a string either in **fixed-length format or in variable-length format** as shown in Figure 13.4.

4.2.1 Fixed-length strings

- ✓ When storing a string in a fixed-length format, you need to **specify an appropriate size** for the string variable. If the **size is too small**, then you will **not be able to store all the elements in the string.** On the other hand, if the **string size is large**, then **unnecessarily memory space will be wasted.**

4.2.2 Variable-length strings

- ✓ A better option is to use a variable length format in which the **string can be expanded or contracted to accommodate the elements in it.** For example, if you declare a string variable to store the name of a student. If a student has a long name of say 20 characters, then the string can be expanded to accommodate 20 characters. On the other hand, a student name has only 5 characters, then the string variable can be contracted to store only 5 characters. However, to use a variable-length string format you need a technique to indicate the end of elements that are a part of the string. This can be done either by using **length-controlled string or a delimiter.**

  1. Length-controlled strings: In a length-controlled string, you need to specify the number of characters in the string.

  2. Delimited strings: In this format, the string is ended with a delimiter such as comma, semicolon, colon, dash, null character etc. The delimiter is then used to identify the end of the string.

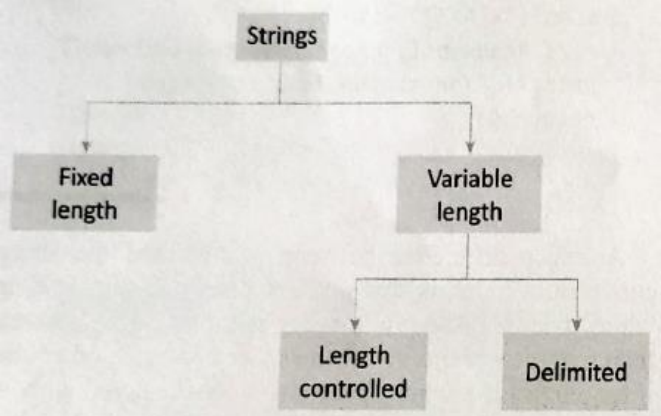


Figure 13.4 String taxonomy

4.3 Operations on Strings

1. Finding Length of a String

- ✓ **The number of characters in a string constitutes the length of the string.** For example, LENGTH("C PROGRAMMING IS FUN") will return 20. Note that even blank spaces are counted as characters in the string.
- ✓ Figure 4.3 shows an algorithm that calculates the length of a string. In this algorithm, I is used as an index for traversing string STR. To traverse each and every character of STR, we increment the value of I. Once we encounter the null character, the control jumps out of the while loop and the length is initialized with the value of I.

```
Step 1: [INITIALIZE] SET I = 0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:    SET I = I + 1
        [END OF LOOP]
Step 4: SET LENGTH = I
Step 5: END
```

Figure 4.3 Algorithm to calculate the length of a string

2. Converting Characters of a String into Upper Case

- ✓ We have already discussed that in the memory ASCII codes are stored instead of the real values. The **ASCII code for A–Z varies from 65 to 91 and the ASCII code for a–z ranges from 97 to 123**. So, if we have to convert a **lower case character into uppercase, we just need to subtract 32 from the ASCII value of the character.**
- ✓ Figure 4.4 shows an algorithm that converts the lower case characters of a string into upper case. In the algorithm, we initialize I to zero. Using I as the index of STR, we traverse each character of STR from Step 2 to 3. If the character is in lower case, then it is converted into upper case by subtracting 32 from its ASCII value. But if the character is already in upper case, then it is copied into the UPPERSTR string. Finally, when all the characters have been traversed, a null character is appended to UPPERSTR (as done in Step 4).

```
Step 1: [INITIALIZE] SET I=0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:   IF STR[I] >= 'a' AND STR[I] <= '2'
            SET UPPERSTR[I] = STR[I] -32
          ELSE
            SET UPPERSTR[I] = STR[I]
          [END OF IF]
          SET I = I + 1
        [END OF LOOP]
Step 4: SET UPPERSTR[I] = NULL
Step 5: EXIT
```

Figure 4.4 Algorithm to convert characters of a string into upper case

3. Converting Characters of a String into Lower Case

- ✓ If we have to convert an **upper case character into lower case, we need to add 32 to the ASCII value of the character.**
- ✓ Figure 13.8 shows an algorithm that converts the upper case characters of a string into lower case.
- ✓ In the algorithm, we initialize I to zero. Using I as the index of STR, we traverse each character of STR from Step 2 to 3. If the character is in upper case, then it is converted into lower case by adding 32 to its ASCII value. But if the character is already in lower case, then it is copied into the Lowerstr string. Finally, when all the characters have been traversed, a null character is appended to Lowerstr (as done in Step 4).

```
Step 1:  [INITIALIZE] SET I = 0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:       IF STR[I] >= 'A' AND STR[I] <= 'Z'
                    SET Lowerstr[I] = STR [I] + 32
               ELSE
                    SET Lowerstr[I] = STR [I]
               [END OF IF]
               SET I = I + 1
       [END OF LOOP]
Step 4: SET Lowerstr[I] = NULL
Step 5: EXIT
```

Figure 13.8 Algorithm to convert characters of a string into lower case

4. Concatenating Two Strings to Form a New String

- ✓ **If s1 and s2 are two strings, then concatenation operation produces a string which contains characters of s1 followed by the characters of s2.**
- ✓ Figure 13.9 shows an algorithm that concatenates two strings. In this algorithm, we first initialize the tow counters I and J to zero. To concatenate the strings, we have to copy the contents of the first string followed by the contents of the second string in the third string new_str. Steps 2 to 4 copies the

contents of first string in new_str. Likewise Steps 6 to 8 copies the contents of second string in new_str. After the contents have been copied, a null character is appended at the end of new_str.

```
Step 1: [INITIALIZE] I = 0 and J = 0
Step 2: Repeat Steps 3 to 4 while str1 [i]! = NULL
Step 3:          SET new_str[J] = str1[I]
Step 4:          Set I = I+1 and J = J+1
        [END of LOOP]
Step 5: SET I=0
Step 6: Repeat Steps 6 to 7 while str2[i]! = NULL
Step 7:          SET new_str[J] = str2 [I]
Step 8:          Set I = I+1 and J = J+1
        [END of LOOP]
Step 9: SET new_str[J] = NULL
Step 10: EXIT
```

Figure 13.9 Algorithm to concatenate two strings

5. Appending a String to Another String

- ✓ **Appending one string to another string involves copying the contents of the source string at the end of the destination string.**
- ✓ For example, if S1 and S2 are two strings, then appending S1 to S2 means we have to add the contents of S1 to S2. So, S1 is the source string and S2 is the destination string. The appending operation would leave the source string S1 unchanged and the destination string S2 = S2 + S1.
- ✓ Figure 4.5 shows an algorithm that appends two strings. In this algorithm, we first traverse through the destination string to reach its end, i.e., reach the position where a null character is encountered. The characters of the source string are then copied into the destination string starting from that position. Finally, a null character is added to terminate the destination string

```
Step 1: [INITIALIZE] SET I=0 and J=0
Step 2: Repeat Step 3 while DEST_STR[I] != NULL
Step 3:          SET I = I + 1
        [END OF LOOP]
Step 4: Repeat Steps 5 to 7 while SOURCE_STR[J] != NULL
Step 5:          DEST_STR[I] = SOURCE_STR[J]
Step 6:          SET I = I + 1
Step 7:          SET J = J + 1
        [END OF LOOP]
Step 8: SET DEST_STR[I] = NULL
Step 9: EXIT
```

Figure 4.5 Algorithm to append a string to another string

6. Comparing Two Strings

- ✓ **If S1 and S2 are two strings, then comparing the two strings will give either of the following results:**

(a) S1 and S2 are equal

(b) S1>S2, when in dictionary order, S1 will come after S2

(c) S1 <S2, when in dictionary order, S1 precedes S2

- ✓ To compare the two strings, each and every character is compared from both the strings. If all the characters are the same, then the two strings are said to be equal.
- ✓ Figure 4.6 shows an algorithm that compares two strings.
- ✓ In this algorithm, we first check whether the two strings are of same length. If not, then there is no point in moving ahead as it straightaway means that the two strings are not same. However, if the two strings are of the same length, then we compare character by character to check if all the characters

are same. If yes, then variable same is set to 1 else if same = 0, then we check which string precedes the other in dictionary order and print the corresponding message.

```
Step 1: [INITIALIZE] SET I=0, SAME =0
Step 2: SET LEN1 = Length(STR1), LEN2 = Length(STR2)
Step 3: IF LEN1 != LEN2
            Write "Strings Are Not Equal"
        ELSE
            Repeat while I<LEN1
                  IF STR1[I] == STR2[I]
                        SET I = I + 1
                  ELSE
                        Go to Step 4
                  [END OF IF]
            [END OF LOOP]
            IF I = LEN1
                  SET SAME =1
                  Write "Strings are Equal"
            [END OF IF]
Step 4: IF SAME = 0,
            IF STR1[I] > STR2[I]
                  Write "String1 is greater than String2"
            ELSE IF STR1[I] < STR2[I]
                  Write "String2 is greater than String1"
            [END OF IF]
        [END OF IF]
Step 5: EXIT
```

Figure 4.6 Algorithm to compare two strings

7. Reversing a String

- ✓ If S1="HELLO", then reverse of S1="OLLEH". **To reverse a string, we just need to swap the first character with the last, second character with the second last character, and so on.**
- ✓ Figure 4.7 shows an algorithm that reverses a string.
- ✓ In Step 1, I is initialized to zero and J is initialized to the length of the string-1. In Step 2, while loop is executed until all the characters of the string are accessed. In Step 3, we swap the i^{th} character of STR with its j^{th} character. In Step 4, the value of I is incremented and J is decremented to traverse STR in the forward and backward direction respectively.

```
Step 1: [INITIALIZE] SET I=0, J= Length(STR)-1
Step 2: Repeat Steps 3 and 4 while I < J
Step 3:     SWAP(STR(I), STR(J))
Step 4:     SET I = I + 1, J = J - 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 4.7 Algorithm to reverse a string

8. Extracting a Substring from Left

- ✓ **In order to extract a substring from the main string we need to copy the content of the string starting from the first position to the n^{th} position where n is the number of characters to be extracted.**
- ✓ For example, if S1= “Hello World”, then Substr_Left(S1,7)=Hello w.
- ✓ The algorithm for extracting the first n characters from a string is shown below:
  Step 1: [INITIALIZE] SET I = 0
  Step 2: Repeat Step 3 to 4 while STR[I] != NULL AND I<N
  Step 3: SET Substr[I] = STR[I]
  Step 4: SET I = I+1

[END OF LOOP]
Step 5: SET Substr[I] = NULL
Step 6: EXIT

Figure 13.13 Algorithm to extract first n characters from a string

- ✓ In Step 1, we initialize the index variable I with zero. In Step 2, a while loop is executed until all the characters of STR have been accessed and I is less than N. In Step 3, the I^{th} character of STR is copied in the I^{th} character of Substr. In Step 4, the value of I is incremented to access the next character in STR. In Step 5, Substr is appended with a null character.

9. Extracting a Substring from Right

- ✓ **In order to extract a substring from the right side of the main string we need to first calculate the position from the left.**
- ✓ For example, if S1= "Hello World" and we have to copy 7 characters starting from the right then we have to actually start extracting characters from the 4th position. This is calculated by total number o characters - n.
- ✓ For example, if S1= "Hello World", then Substr_Right(S1, 7) = o World
- ✓ Figure 13.14 shows an algorithm that extracts n characters from the right of a string.

Step 1: [INITIALIZE] SET I = 0, J = Length(STR)-N
Step 2: Repeat Step 3 to 4 while STR[J] != NULL
Step 3: SET Substr[I] = STR[J]
Step 4: SET I = I+1, J=J+1
[END OF LOOP]
Step 5: SET Substr[I] = NULL
Step 6: EXIT

Figure 13.14 Algorithm to extract n characters from the right of a string

- ✓ In Step 1, we initialize the index variable I to zero and J to Length (STR)-N so that J points to the character from which the string has to be copied in the substring. In Step 2, a while loop is executed until the null character in STR is accessed. In Step 3, the J^{th} character of STR is copied in the I^{th} character of Substr. In Step 4, the value of I and J are incremented. In Step 5, Substr is appended with a null character.

10. Extracting a Substring from the Middle of a string

- ✓ **To extract a substring from a given string requires information about three things. The main string, the position of the first character of the substring in the given string and the number of characters/length of the substring.**
- ✓ For example, if we have a string,
  str[] = "Welcome to the world of programming"
  then
  SUBSTRING (str, 15, 5) = World
- ✓ Figure 4.8 shows an algorithm that extracts the substring from a middle of a string.

```
Step 1: [INITIALIZE] Set I=M, J=0
Step 2: Repeat Steps 3 to 6
           while STR[I] != NULL and N>0
Step 3:  SET SUBSTR[J] = STR[I]
Step 4:  SET I = I + 1
Step 5:  SET J = J + 1
Step 6:  SET N = N - 1
        [END OF LOOP]
Step 7: SET SUBSTR[J] = NULL
Step 8: EXIT
```

Figure 4.8 Algorithm to extract a substring from the middle of a string

- ✓ In this algorithm, we initialize a loop counter I to M. i.e., the position from which the characters have to be copied. Steps 3 to 6 are repeated until N characters have been copied. With every character copied, we decrement the value of N. The characters of the string are copied into a string called substr. At the end a null character is appended to substr to terminate the string.

11. Inserting a String in Another String

- ✓ **The insertion operation inserts a string S in the main text T at the k^{th} position. The general syntax of this operation is INSERT(text, position, string).**
- ✓ For example, INSERT("XYZXYZ",3, "AAA") = "XYZAAAXYZ" Figure 4.9 shows an algorithm to insert a string in a given text at the specified position. This algorithm first initializes the indices into the string to zero. From Steps 3 to 5, the contents of NEW_STR are built. If I is exactly equal to the position at which the substring has to be inserted, then the inner loop copies the contents of the substring into NEW_STR. Otherwise, the contents of the text are copied into it.

12. Indexing

- ✓ **This operation returns the position in the string where the string pattern first occurs.**
- ✓ For example, INDEX("Welcome to the world of programming", "world") = 15
- ✓ However, if the pattern does not exist in the string, the INDEX function returns 0.

13. Deleting a string from the Main String

- ✓ **The deletion operation deletes a substring from a given text.**
- ✓ We can write it as DELETE(text, position, length).
- ✓ For example, DELETE("ABCDXXXABCD", 4, 3) = "ABCDABCD"

14. Replacing a Pattern with Another Pattern in a String

- ✓ **The replacement operation is used to replace the pattern P1 by another pattern P2.**
- ✓ This is done by writing REPLACE(text, pattern1 , pattern2).
- ✓ For example, ("AAABBBCCC", "BBB", "X") = AAAXCCC
  ("AAABBBCCC", "X", "YYY")= AAABBBCC

4.3 Miscellaneous String and Character Functions

4.3.1 Character Manipulation Functions

- ✓ Table 8.1 illustrates some character functions contained in ctype.h.

Prototype	Function description
`int isblank( int c );`	Returns a true value if c is a *blank character* that separates words in a line of text and 0 (false) otherwise. [*Note:* This function is not available in Microsoft Visual C++.]
`int isdigit( int c );`	Returns a true value if c is a *digit* and 0 (false) otherwise.
`int isalpha( int c );`	Returns a true value if c is a *letter* and 0 otherwise.
`int isalnum( int c );`	Returns a true value if c is a *digit* or a *letter* and 0 otherwise.
`int isxdigit( int c );`	Returns a true value if c is a *hexadecimal digit character* and 0 otherwise. (See Appendix C for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c );`	Returns a true value if c is a *lowercase letter* and 0 otherwise.
`int isupper( int c );`	Returns a true value if c is an *uppercase letter* and 0 otherwise.
`int tolower( int c );`	If c is an *uppercase letter*, `tolower` returns c as a *lowercase letter*. Otherwise, `tolower` returns the argument unchanged.

`int toupper( int c );`	If `c` is a *lowercase letter*, `toupper` returns `c` as an *uppercase letter*. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c );`	Returns a true value if `c` is a *whitespace character*—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`)—and 0 otherwise.
`int iscntrl( int c );`	Returns a true value if `c` is a *control character* and 0 otherwise.
`int ispunct( int c );`	Returns a true value if `c` is a *printing character other than a space, a digit, or a letter* and returns 0 otherwise.
`int isprint( int c );`	Returns a true value if `c` is a *printing character including a space* and returns 0 otherwise.
`int isgraph( int c );`	Returns a true value if `c` is a *printing character other than a space* and returns 0 otherwise.

Fig. 8.1 | Character-handling library (<ctype.h>) functions. (Part 2 of 2.)

4.3.2 String Manipulation Functions

✓ The standard library 'string.h' contains many functions for the string manipulation.

1. strlen(str): The length of a string

✓ The 'strlen()' function can be used to find the length of the string in bytes.

✓ This function calculates the length of the string excluding the 'null character'. i.e., the function returns the number of characters in the string.

Syntax: size_t strlen(const char *str);

Ex: Write a C program to demonstrate the usage of strlen().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char str[] ="HELLO WORLD!";
        printf("\n The length of the string is: %d", strlen(str));
}
```

Output:

The length of the string is: 5

2. strcpy ( ): String Copy

✓ The 'strcpy()' function copies the contents of source string str2 to destination string str1 including '\0'.

✓ The strcpy() function copies characters from the source string to the destination string until it finds null character. It returns the argument str1.

Syntax: char * strcpy(char *str1, const char *str2);

Where,

str1: it is the destination string

str2: it is the source string.

Ex: Write a C program to demonstrate the usage of strcpy().

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

void main()
```

```
{
        char str1[10], str2[10]= "JAIN";
        strcpy(str1,str2);
        printf("The Source String=%s\n The Destination String=%s", str1,str2);
}
```

Output:

The Source String= JAIN
The Destination String= JAIN

3. strncpy(str1,str2,n): String Number Copy

- ✓ 'strncpy()' function is used to copy 'n' characters from the source string str2 to the destination string str1.

Syntax : char *strncpy(char *str1, const char *str2, size_t n);

Where,
str1: it is the destination string.
str2: it is the source string.
n: n is the number of characters to be copied into destination string.

Ex: Write a C program to demonstrate the usage of strncpy().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[10], str2[10]= "Computer";
        strncpy(str1,str2,3);
        printf ("The Source String=%s\n The Destination String=%s", str1,str2);
}
```

Output:

The Source String= Computer
The Destination String= Com

4. strcat(str1,str2): String Concatenate(Joining two strings together)

- ✓ The strcat function appends the string pointed to by str2 to the end of the string pointed to by str1.
- ✓ The 'strcat()' function is used to concatenate or join the two strings.
- ✓ The 'strcat()' function copies all the characters of string str2 to the end of string str1.
- ✓ The NULL character of string str1 is replaced by the first character of str2.

Syntax: char *strcat(char *str1, const char *str2);

Where,
str1: It is the first string
str2: It is the second string

Ex: Write a C program to demonstrate the usage of strcat().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[15]= "Good";
        char str2[15]= "Morning";
        strcat(str1,str2);
        printf("The concatenated String=%s",str1);
}
```

Output:
The concatenated String=GoodMorning.

5. strncat(str1,str2,n)- String Number Concatenate

- ✓ The 'strncat()' function is used to concatenate or join n characters of string str2 to the end of string str1.

Syntax: char *strcat(char *str1, const char *str2, size_t n);
Where,
str1: It is the first string
str2: It is the second string
n: It is the number of characters of string s2 to be concatenated.

Ex: Write a C program to demonstrate the usage of strncat().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[15]= "Good";
        char str2[15]= "Morning";
        strncat(str1,str2,4);
        printf("The concatenated String=%s",str1);
}
```

Output:
The concatenated String=GoodMorn.

6. strcmp(str1,str22): String Compare

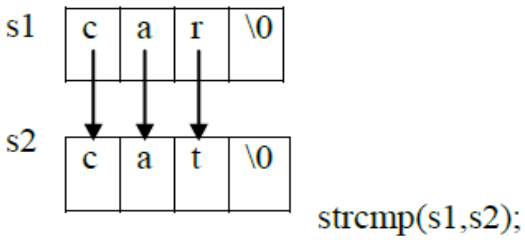
- ✓ This function is used to compare two strings.
- ✓ The comparison starts with first character of each string. The comparison continues till the corresponding characters differ or until the end of the character is reached.
- ✓ The following values are returned after comparison:

1. If two strings are equal, the function returns 0.
2. If str1 is greater than str2, a positive value is returned.
3. If str1 is less than str2, then the function returns a negative value.

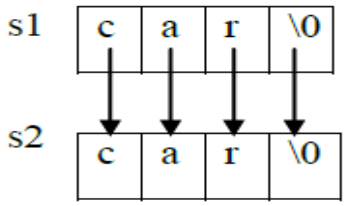
Syntax: int strcmp(const char *str1, const char *str2);
Where,
str1: It is the first string.
str2: It is the second string.

Ex:

- ✓ "car" and "cat" are different strings. The characters 'r' and 't' have different ASCII values. It returns negative value since ASCII value of 'r' is less than the ASCII value of 't'.

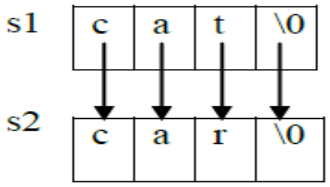


✓ “car” and “car” are same, the function returns 0.



strcmp(s1,s2);

✓ “cat” and “car” are different strings. The characters ‘t’ and ‘r’ have different ASCII values. It returns positive value since ASCII value of ‘t’ is greater than the ASCII value of ‘r’.



strcmp(s1,s2);

Ex: Write a C program to demonstrate the usage of strcmp().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[10]=”Hello”;
        char str2[10]=”Hey”;
        if(strcmp(str1,str2)==0)
                printf(“The two strings are identical”);
        else
                printf(“The two strings are not identical”);
}
```

Output:

The two strings are not identical

7. strncmp(str1,str2,n): String Number Compare

✓ This function is used to compare first n number of characters in two strings.
✓ The comparison starts with first character of each string. The comparison continues till the corresponding characters differ or until the end of the character is reached or specified numbers of characters have been tested.
✓ The following values are returned after comparison:

1. If two strings are equal, the function returns 0.
2. If str1 is greater than str2, a positive value is returned.
3. If str1 is less than str2, then the function returns a negative value.

Syntax: int strcmp(const char *str1, const char *str2, size_t n);

Where,
str1: It is the first string.
str2: It is the second string.
n: It is the number of characters to be compared.

Ex: Write a C program to demonstrate the usage of strcmp().

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

```
#include<string.h>
void main()
{
        char str1[10]=”Hello”;
        char str2[10]=”Hey”;
        if(strncmp(str1,str2,2)==0)
                printf(“The first two characters in the strings are identical”);
        else
                printf(“The first two characters in the strings are not identical”);
}
```

Output:

The first two characters in the strings are identical

8. strchr()

- ✓ It takes a string and a character as input and finds out the first occurrence of the given character in the string pointed to by the argument str.
- ✓ It will return the pointer to the first occurrence of that character; if found otherwise, return Null.

Syntax:

char *strchr(const char *str, int c);

Ex: Write a C program to demonstrate the usage of strchr().

```
#include <stdio.h>
#include <string.h>
void  main()
{
        char string1[30] = "I love to write.";
        char *pos;
        pos= strchr(string1, 'w');
         if(pos)
                printf(“w is found at position %d”, pos);
        else
                printf(“w is not found”);
}
```

Output:

w is found at position 10

9. strrchr()

- ✓ It takes a string and a character as input and finds out the last occurrence of a given character in that string.
- ✓ It will return the pointer to the last occurrence of that character if found otherwise, return Null.

Syntax:

char *strrchr(const char *str, int c);

Ex: Write a C program to demonstrate the usage of strrchr().

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

```
#include <string.h>
 void main()
{
        char string1[30] = “Programming in C.";
        char *pos;
        pos= strrchr(string1, ‘n');
        if(pos)
                printf(“The last position of n is %d”, pos);
        else
                printf(“n is not found”);
}
```

Output:

The last position of n is 13.

10. strspn()

✓ The function returns the index of the first character in str1 that dosen’t match any character in str2.

Syntax:

size_t strspn(const char *str1, const char *str2);

Ex: Write a C program to demonstrate the usage of strspn().

```
#include <stdio.h>
#include <string.h>
main()
{
        char str1[] = “HAPPY BIRTHDAY TO YOU";
        char str2[]=“HAPPY BIRTHDAY JOE”;
        printf(“The position of the first character in str2 that does not
              match with that in str1 is %d”, strspn(str1,str2));
}
```

Output:

The position of the first character in str2 that does not match with that in str1 is 15

11. strcspn()

✓ The function returns the index of the first character in str1 that matches any of the characters in str2.

Syntax:

size_t strcspn(const char *str1, const char *str2);

Ex: Write a C program to demonstrate the usage of strcspn().

```
#include <stdio.h>
#include <string.h>
void main()
{
        char str1[] = “PROGRAMMING IN C";
        char str2[]=“IN”;
        printf(“The position of the first character in str2 that match with
              that in str1 is %d”, strcspn(str1,str2));
}
```

Output:

The position of the first character in str2 that match with that in str1 is 8

12. strpbrk()

- ✓ The function strpbrk() returns a pointer to the first occurrence in str1 of any character in str2, or NULL if none are present.
- ✓ The only difference between strpbrk() and strcspn() is that strcspn() returns the index of the character and strpbrk() returns a pointer to the first occurrence of a character in str2.

Syntax:

char *strpbrk(const char *str1, const char *str2);

Ex: Write a C program to demonstrate the usage of strpbrk().

```
#include <stdio.h>
#include <string.h>
void main()
{
        char str1[] = “PROGRAMMING IN C";
        char str2[]=“AB”;
        char *ptr = strpbrk(str1,str2);
        if(ptr== NULL)
                printf(“\n No character matches in the two strings”);
        else
                printf(“\n  character in str2  matches with that in str1”);
}
```

Output:

character in str2 matches with that in str1

13. strtok()

- ✓ The strtok() function is used to isolate sequential tokens in a null –terminated string, str.
- ✓ It returns a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a null character.
- ✓ When all tokens are left , a null pointer is returned.

Syntax:

char *strtok(char *str1, const char *delimiter);

Ex: Write a C program to demonstrate the usage of strtok().

```
#include <stdio.h>
#include <string.h>
void main()
{
        char str[] = "Hello, to, the, World of, Programming";
        char delim[] =",";
        char *result;
        result = strtok(str,delim);
        while(result!= NULL)
        {
                printf(“\n %s", result);
                result = strtok(NULL,delim);
        }
}
```

Output:
Hello
to
the
World of
Programming

4.3.3 String Manipulation functions present in <stdlib.h>

1. strtol()

- ✓ The strtol() function converts the string pointed by str to a long value.
- ✓ The function skips the leading white space characters and stops when it encounters the first non-numeric character.
- ✓ It stores the address of the first invalid character in str in *end.
- ✓ You may pass NULL instead of *end if you do not want to store any invalid characters anywhere.
- ✓ Third argument base specifies whether the number is Hexadecimal, binary, octal or decimal representation.

Syntax:
long strtol(const char *str, char **end, int base);

Ex: Write a C program to demonstrate the usage of strtol().

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        long num;
        num = strtol("12345 Decimal Value" , NULL, 10);
        printf("%ld", num);
        num = strtol("65432 Octal Value" , NULL, 8);
        printf("%ld", num);
        num = strtol("10110101 Binary Value" , NULL, 2);
        printf("%ld", num);
        num = strtol("A7CB4 Hexadecimal Value" , NULL, 16);
        printf("%ld", num);
}
```

Output:
12345
27418
181
687284

2. strtod()

- ✓ The function accepts a string str that has an optional plus('+') or minus sign('-') followed by either:
- ✓ A decimal number containing a sequence of decimal digits optionally consisting of a decimal point, or
- ✓ A Hexadecimal number consisting of a "OX" or "Ox" followed by a sequence of hexadecimal digits optionally containing a decimal point.
- ✓ In both the cases the number should be optionally followed by an exponent ('E' or 'e' for decimal constants and 'P' or 'p' for hexadecimal constants).

Syntax:
double strtod(const char *str, char **end);

Ex: Write a C program to demonstrate the usage of strtod().

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
        double sum;
        num = strtod("123.345abcdefg", NULL);
        printf("%lf", num);
}
```

Output:
123.345000

3. atoi()

- ✓ The atoi() function converts a given string passed to it as an argument into an integer.
- ✓ The function returns that integer to the calling function.
- ✓ However, the string should start with a number.
- ✓ The function will stop reading from the string as soon as it encounters a non-numerical character.

Syntax:
int atoi(const char *str);

Example:
i = atoi("123.456");
Result: i=123.

4. atof()

- ✓ This function converts the string that it accepts as an argument into a double value and then returns that to the calling function.
- ✓ The string can be terminated with any non-numerical character other than "E" or "e".

Syntax:
double atof(const char *str);

Example:
x = atof("12.39 is the answer");
Result: x=12.39

5. atol()

- ✓ This function converts the string into a long int value.
- ✓ This function will read from the string until it finds any character that should not be in long.

Syntax:
long atol(const char *str);

Example:
x= atol("12345.6789");
Result: x = 12345L.

4.4 Arrays of Strings

✓ An array of strings is declared as

<data_type> <array_name> [row_size][column_size];

Here, the first index row_size will specify how many strings are needed and the second index column_size will specify the length of every individual string.

Ex: char names[20][30];

✓ So here, we will allocate space for 20 names where each name can be a maximum 30 characters long.

✓ Let us see the memory representation of an array of strings. If we have an array declared as char name[5][10] = {"Ram", "Mohan", "Shyam", "Hari", "Gopal"};

Then in the memory, the array will be stored as shown in Fig. 4.13.

name[0]	R	A	M	'\0'						
name[1]	M	O	H	A	N	'\0'				
name[2]	S	H	Y	A	M	'\0'				
name[3]	H	A	R	I	'\0'					
name[4]	G	O	P	A	L	'\0'				

Figure 4.13 Memory representation of a 2D character array

✓ By declaring the array names, we allocate 50 bytes. But the actual memory occupied is 27 bytes. Thus, we see that about half of the memory allocated is wasted.

✓ Figure 4.14 shows an algorithm to process individual string from an array of strings. In Step 1, we initialize the index variable I to zero. In Step 2, a while loop is executed until all the strings in the array are accessed. In Step 3, each individual string is processed.

```
Step 1: [INITIALIZE] SET I=0
Step 2: Repeat Step 3 while I< N
Step 3:     Apply Process to NAMES[I]
         [END OF LOOP]
Step 4: EXIT
```

Figure 4.14 Algorithm to process individual string from an array of strings

Ex: Write a C Program to Read and Print the Names of N Students of a Class.

```
#include<stdio.h>
void main()
{
        char names[5][10];
        int i, n;
        printf("\n Enter the number of students : ");
        scanf("%d", &n);
        printf("\n Enter the names of students :");
        for(i=0;i<n;i++)
        {
                scanf("%s",names[i]);
        }
        printf("\n Names of the students are : \n");
        for(i=0;i<n;i++)
                puts(names[i]);
}
```

Module 4

Pointers

4.6 Introduction

- ✓ **"A pointer is a variable that holds the address of another variable". or**
- ✓ **A pointer is a variable that contains the memory location of another variable. Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element.**

Applications of pointer

- ✓ Pointers are used to pass information back and forth between functions.
- ✓ Pointers enable the programmers to return multiple data items from a function via function arguments.
- ✓ Pointers provide an alternate way to access the individual elements of an array.
- ✓ Pointers are used to pass arrays and strings as function arguments.
- ✓ Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.
- ✓ Pointers are used for the dynamic memory allocation of a variable.

4.7 Declaring Pointer Variables

- ✓ Pointer provides **access** to a variable by **using the address of that variable.**
- ✓ A pointer variable is therefore a **variable that stores the address of another variable.**
- ✓ The general syntax of declaring pointer variables can be given as below.

data_type *ptr_name;

Here, **data_type:** is the data type of the value that the pointer will point to. It can be int, float, char etc.

Asterisk (*): It tells the compiler that we are declaring a pointer variable.
pointer_variable_name: It is the name of the pointer variable.

Example:

1. int *ptr; // declares a pointer variable ptr of integer type.
2. float *temp; // declares a pointer variable temp of floating type.

Operators used with Pointers:

The two basic operators used with pointers are:

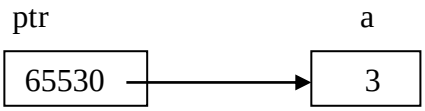
i. **The Address of operator (&):** By using the address of (&) operator, we can determine the address of the variable.

ii. **The Indirection operator (*):** It gives the value stored at a particular address.

Example:

```
int a=3;
int *ptr;
ptr=&a;
```

Memory layout:



Address: 65530

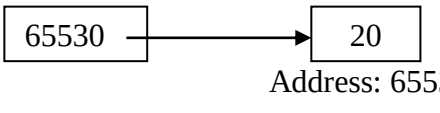
✓ 'ptr = &a' copies the address of 'a' to the pointer variable 'ptr'.

Example Program: Write a C program to print value and address of the variable using pointers.

```
#include<stdio.h>
void main ()
{
    int a=20, *ptr;
    ptr = &a;                          //ptr1 is a pointer to variable a
    printf("The address of a=%d and value of a=%d\n",ptr,*ptr);
}
```

Output:
The address of a=65530 and value of a=20

Memory layout:



Address: 65530

Initializing a Pointer Variable

✓ We can initialize the pointer variables by assigning the address of other variable to them. However these variables must be declared in the program.

Syntax

```
data_type * ptr_name = address_of_variable;
```

where,

data_type:.It can be int, float, char etc.

Asterisk (*): It tells the compiler that we are declaring a pointer variable.

ptr_name: It is the name of the pointer variable.

address_of_variable: Itis the address of another variable.

Example:

```
int a;
int *ptr;
ptr=&a;
  or
int a;
int *ptr=&a;
```

Both are equivalent.

✓ We can dereference a pointer, i.e., **refer to the value of the variable to which it points, by using unary '*' operator** (also known as indirection operator) as *pnum, i.e., *pnum = 10, since 10 is value of x. Therefore, * is equivalent to writing value at address. Look at the code below which shows the use of pointer variable.

```
#include <stdio.h>
void main()
{
        int num, *pnum;
        pnum = &num;
```

```
        printf("Enter the number : ");
        scanf("%d", &num);
        printf("\n The number that was entered is : %d", *pnum);
}
```

Output:

Enter the number : 10
The number that was entered is : 10
What will be the value of *(&num)? It is equivalent to simply writing num.

4.8 Types of Pointer

4.8.1 Null Pointers

- ✓ We have studied that a pointer variable is a pointer to a variable of some data type.
- ✓ However, in some cases, we may prefer to have a **null pointer which is a special pointer value and does not point to any value.**
- ✓ This means that a **null pointer does not point to any valid memory address.**
- ✓ To **declare a null pointer**, you may use the **predefined constant NULL** which is defined in several standard header files including <stdio.h>, <stdlib.h> , and <string.h>.
- ✓ After including any of these files in your program, you can write

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

- ✓ You can always check whether a given pointer variable stores the address of some variable or contains NULL by writing,

```
if (ptr == NULL)
{
    Statement block;
}
```

- ✓ You may also initialize a pointer as a null pointer by using the constant 0

```
int *ptr, ptr = 0;
```

- ✓ This is a valid statement in C as NULL is a preprocessor macro, which typically has the value or replacement text 0. However, to avoid ambiguity, it is always better to use NULL to declare a null pointer. A function that returns pointer values can return a null pointer when it is unable to perform its task.
- ✓ Null pointers are **used in situations where one of the pointers in the program points to different locations at different times.**

4.8.2 Generic Pointers

- ✓ **A generic pointer is a pointer variable that has void as its data type.**
- ✓ **The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type.**
- ✓ It is declared like a normal pointer variable but using the **void keyword** as the pointer's data type. For example,

```
void *ptr;
```

- ✓ In C, since you cannot have a variable of type void, the void pointer will therefore not point to any data and, thus, **cannot be dereferenced.** You need to **cast a void pointer to another kind of pointer before using it.**
- ✓ Generic pointers are often used when you want a pointer to point to data of different types at different times. For example,

```
#include <stdio.h>
void main()
{
    int x=10;
    char ch = 'A';
    void *gp;
    gp = &x;
    printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
    gp = &ch;
    printf("\n Generic pointer now points to the character= %c", *(char*)gp);
}
```

Output:
Generic pointer points to the integer value = 10
Generic pointer now points to the character = A

4.9 Passing Arguments to Function Using Pointers

- ✓ Using **call-by-value method**, it is **impossible to modify the actual parameters** when you pass them to a function. Furthermore, the **incoming arguments** to a function are treated as **local variables** in the function and those **local variables get a copy of the values** passed from their calling function.
- ✓ **Pointer provides a mechanism to modify data declared in one function using code written in another function.** In other words: If data is declared in func1() and we write code in func2() that modifies the data in func1(), then we must pass the addresses of the variables to func2().
- ✓ **The calling function sends the addresses of the variables and the called function declares those incoming arguments as pointers.** In order to modify the variables sent by the calling function, the called function must dereference the pointers that were passed to it. Thus, passing pointers to a function avoids the overhead of copying data from one function to another.
- ✓ Hence, to use pointers for passing arguments to a function, the programmer must do the following:
    - Declare the function parameters as pointers.
    - Use the referenced pointers in the function body.
    - Pass the addresses as the actual argument when the function is called.

1. Write a C program to add two numbers using call by reference.

```
#include<stdio.h>

int add (int *a,int *b)
{
        int  sum;
        sum = *a + *b;
        return sum;
}

void main()
{
        int a,b, res;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        res = add(&a,&b);
        printf("result =%d\n", res);
}
```

Output:
Enter the values of a and b: 4 5
result =9

2. Write a C program to swap two numbers using call by reference.

```
#include<stdio.h>
void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
void main()
{
        int a,b;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        printf("Before swapping: a=%d\tb=%d", a, b);
        swap(&a,&b);
        printf("After swapping: a=%d\tb=%d", a, b)
}
```

```
Enter the values of a and b: 10 20
Before swapping: a=10        b=20
After swapping: a=20         b=10
```

Output:

Enter the values of a and b: 10 20
Before swapping: a=10 b=20
After swapping: a=20 b=10

Module 5
Structure, Union, and Enumerated Data Type

5.1 Introduction

- ✓ **Structure** is basically a **user-defined data type** that can store related information (even of different data types) together.
- ✓ The major difference between a structure and an array is that an array can store only information of same data type.
- ✓ **A structure is a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure**.
- ✓ **"A Structure is a user defined data type, which is used to store the values of different data types together under the same name".**

5.1.1 Structure Declaration

- ✓ A structure is declared using the keyword **struct** followed by the structure name.
- ✓ All the variables of the structure are declared within the structure.
- ✓ A structure type is generally declared by using the following syntax:

```
struct struct–name
{
        data_type var–name;
        data_type var–name;
        ..............
};
```

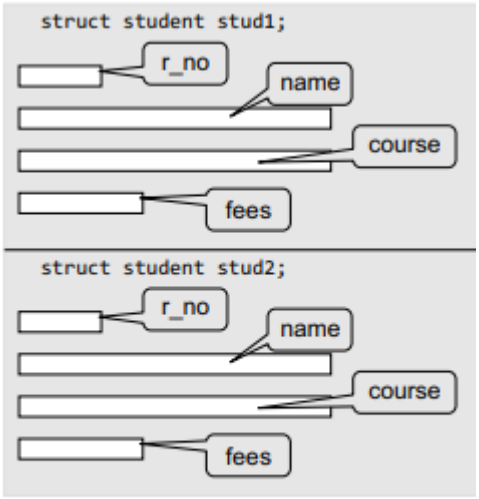


Figure 5.1 Memory allocation for a structure variable

For example:

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

- ✓ A variable of structure student can be defined by writing:
  struct student stud1;
  struct student is a data type and **stud1** is a variable.
- ✓ In the following syntax, the variables are declared at the time of structure declaration.

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
} stud1, stud2;
```

stud1 and stud2 of the structure student.

5.1.2 Typedef Declarations

- ✓ The typedef (derived from type definition**) keyword enables the programmer to create a new data type name by using an existing data type alternate name is given to a known data type.**
- ✓ The general syntax of using the typedef keyword is given as:

  typedef existing_data_type new_data_type;

  For example: typedef int INTEGER;

  then INTEGER is the new name of data type int.
- ✓ To declare variables using the new data type name, precede the variable name with the data type name (new).
- ✓ Therefore, to define an integer variable, INTEGER num=5;

For example, consider the following declaration:

```
typedef struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

5.1.3 Initialization of Structures

- ✓ Initializing a structure means **assigning some constants to the members of the structure.**
- ✓ When the **user does not explicitly initialize the structure, then C automatically does it.**
- ✓ For **int and float members, the values are initialized to zero, and char and string members are initialized to '\0' by default.**
- ✓ The initializers are enclosed in braces and are separated by commas.
- ✓ The general syntax to initialize a structure variable is as follows:

```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .....................
}struct_var = {constant1, constant2, constant3,...};
                or
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
         .....................
};
struct struct_name struct_var = {constant1, constant2, constant 3,...};
```

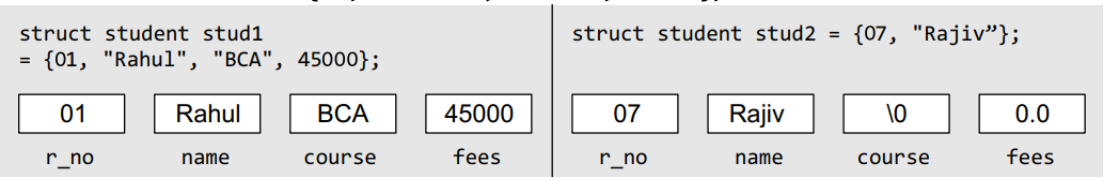
- ✓ For example, we can initialize a student structure by writing,

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
}stud1 = {01, "Rahul", "BCA", 45000};
```

Or,

by writing,

struct student stud1 = {01, "Rahul", "BCA", 45000};



5.1.4 Accessing the Members of a Structure

- ✓ A structure member variable is generally accessed using a **'.' (dot) operator**. The syntax of accessing a structure or a member of a structure can be given as:

  struct_var. member_name

  For example, to assign values to the individual data members of the structure variable studl, we may write

  stud1.r_no = 01;

  stud1.name = "Rahul";

  stud1.course = "BCA";

  stud1.fees = 45000;

- ✓ To input values for data members of the structure variable stud1, we may write

  scanf("%d", &stud1.r_no);

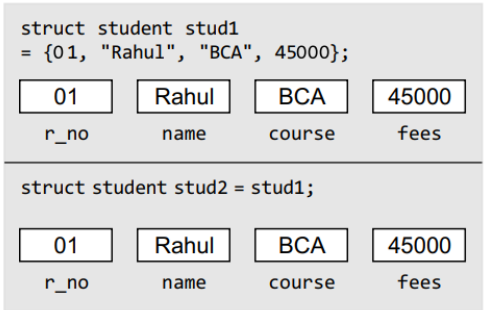
  scanf("%s", stud1.name);

- ✓ Similarly, to print the values of structure variable stud1, we may write

  printf("%s", stud1.course);

  printf("%f", stud1.fees);

5.1.5 Copying and Comparing Structures



- ✓ We can assign a structure to another structure of the same type:
- ✓ Then to assign one structure variable to another, we will write

  stud2 = stud1;

- ✓ C does not permit comparison of one structure variable with another. However, individual members of one structure can be compared with individual members of another structure.
- ✓ For example, to compare the fees of two students, we will write

  if(stud1.fees > stud2.fees) //to check if fees of stud1 is greater than stud2

5.1.6 Finding the Size of the structures

- ✓ There are two different ways through which we can find the number of bytes a structure will occupy in the memory.

1. Simple Addition

- ✓ **In this techniques, make a list of all data types and add the memory required by each.**
- ✓ **Consider a simple structure of an employee**

```
struct Employee
{
        int emp_id;
        char name[20];
        double salary;
        char designation[20];
        float experience;
};
```

Size = size of emp_id+ size of name + size of salary + size of designation + size of experience
Size of emp_id = 2
Size of name =20* size of character
Size of salary = 8
Size of designation = 20 * Size of character
Size of experience = 4
Therefore, Size = 2 + 20*1 +8 +20 *1+4
= 2 +20+8+20+4
= 54bytes

2. Using sizeof operator

- ✓ The **sizeof** operator is used to calculate the **size of a data type, variable, or an expression.**
- ✓ This operator can be used as follows:

sizeof(struct_name);

Ex:

```
#include<stdio.h>
struct employee
{
        int emp_id;
        char name[10];
        double salary;
        char designation [20];
        float experience;
};

void main()
{
        struct employee e;
        printf("%d", sizeof(e));
}
```

Example: C program to read and display the student details using structures.

```
#include<stdio.h>
struct student
{
    int rnum;
    char name[20];
    int marks;
}s[60];

void main()
{
    int i,n;
    printf("Enter the number of students");
```

```
    scanf("%d",&n);

    printf("\nEnter the roll number, Name , Marks\n");
    for(i=1;i<=n;i++)
    {
            printf("\nStudent %d details\n",i);
            scanf("%d",&s[i].rnum);
            scanf("%s",s[i].name);
            scanf("%d",&s[i].marks);
     }
    printf("\nStudent Details are:");
    printf("\nRoll_number\tName\tMarks");
    for(i=1;i<=n;i++)
            printf("\n%d\t\t%s\t%d\n",s[i].rnum,s[i].name,s[i].marks);

  }
```

Output:

```
Enter the number of students: 5
Enter the roll number,
Name, Marks and Grade of
Student 1 details
14
Dhavan
89
Student 2 details
15
Karan
55
Student 3 details
11
Deepa
45
Student 4 details
12
Lakshmi
35
Student 5 details
10
Soma
68
```

Student Details are:

Roll_number	Name	Marks
14	Dhavan	89
15	Karan	55
11	Deepa	45
12	Lakshmi	35
10	Soma	68

5.2 Structures and Functions

✓ A function may access the members of a structure in three ways.

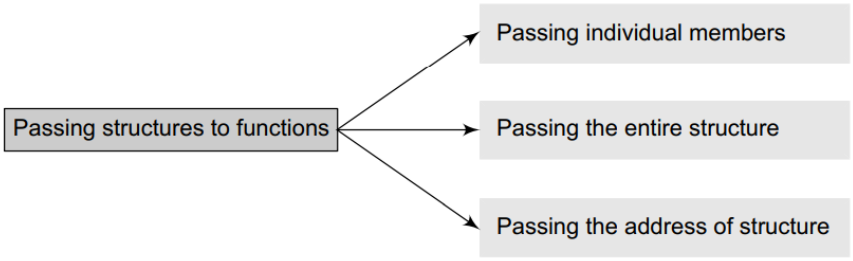


Figure 5.4 Different ways of passing structures to functions

1. Passing Individual Members

✓ To pass any individual member of a structure to a function, we must use the direct selection operator to refer to the individual members.

```
#include<stdio.h>
typedef struct
{
        int x;
        int y;
}POINT;

void display(int, int);

void main()
{
        POINT p1 = {2, 3};
        display(p1.x, p1.y);
}

void display(int a, int b)
{
        printf(" The coordinates of the point are: %d %d", a, b);
}
```

Output: The coordinates of the point are: 2 3

2. Passing the Entire Structure

✓ The entire structure can be passed as a function argument.

✓ When a structure is passed as an argument, it is passed using the call by value method, i.e., a copy of each member of the structure is made.

✓ The general syntax for passing a structure to a function and returning a structure can be given as,

struct struct_name func_name(struct struct_name struct_var);

```
#include <stdio.h>
typedef struct
{
        int x;
        int y;
}POINT;

void display(POINT);
```

```
void main()
{
        POINT p1 = {2, 3};
        display(p1);
}

void display(POINT p)
{
        printf("The coordinates of the point are: %d %d", p.x, p.y);
}
```

3. Passing Structures through Pointers

✓ The syntax to declare a pointer to a structure can be given as,

```
struct struct_name
{
      data_type member_name1;
      data_type member_name2;
      data_type member_name3;
      ......................
}*ptr;
```

or,

```
struct struct_name *ptr;
```

✓ For the student structure, we can declare a pointer variable by writing

struct student *ptr_stud, stud;

And to assign the address, we will write

ptr_stud = &stud;

✓ To access the members of a structure, we can write

/* get the structure, then select a member */

(*ptr_stud).roll_no;

This operator is known as 'pointing-to' operator (->). It can be used as:

ptr_stud ->roll_no = 01;

Example: Write a program using pointer to structure to initialize the members in the structure.

```
#include<stdio.h>
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};

void main()
{
        struct student stud1,  *ptr_stud1;
        ptr_stud1 = &stud1;
        ptr_stud1->r_no = 01;
        strcpy(ptr_stud1->name, "Rahul");
        strcpy(ptr_stud1->course, "BCA");
```

```
        ptr_stud1->fees = 45000;
        printf("\n DETAILS OF STUDENT");
        printf("\n ------------------------------------------");
        printf("\n ROLL NUMBER = %d", ptr_stud1->r_no);
        printf("\n NAME  = %s", ptr_stud1->name);
        printf("\n COURSE  = %s", ptr_stud1->course);
        printf("\n FEES = %f", ptr_stud1->fees);
    }
```

5.3 Union

- ✓ Like structure, **a union is a collection of variables of different data types.** The only difference between a structure and a union is that in case of unions, **you can only store information in one field at any one time.**
- ✓ To better understand union, think of it as a chunk of memory that is used to store variables of different types. When a new value is assigned to a field, the existing data is replaced with the new data.

5.3.1 Declaring a Union

- ✓ The syntax for declaring a union is same as that of declaring a structure.

```
union union-name
{
        data_type var-name;
        data_type var-name;
        …….
};
```

- ✓ Again, the typedef keyword can be used to simplify the declaration of union variables.
- ✓ The most important thing to remember about a union is that the size of an **union is the size of its largest field**. This is because a sufficient number of bytes must be reserved to store the largest sized field.

5.3.2 Accessing a Member of a Union

- ✓ **A member of a union can be accessed using the same syntax as that of a structure.**
- ✓ To access the fields of a union, use the **dot operator(.).**
- ✓ That is the union variable name followed by the dot operator followed by the member name.

5.3.3 Initializing Unions

- ✓ It is an error to initialize any other union member except the first member.
- ✓ A striking difference between a structure and a union is that in case of a union, the fields share the same memory space, so fresh data replaces any existing data. Look at the code given below and observe the difference between a structure and union when their fields are to be initialized.

```
#include<stdio.h>
typedef struct POINT1
{
        int x, y;
};
typedef union POINT2
{
        int x;
        int y;
};
```

```
void main()
{
        POINT1 P1 = {2,3};
        // POINT2 P2 ={4,5}; Illegal with union
        POINT2 P2;
        P2. x = 4;
        P2.y = 5;
        printf("\n The co-ordinates of P1 are %d and %d", P1.x, P1.y);
        printf("\n The co-ordinates of P2 are %d and %d", P2.x, P2.y);
}
```

Output:

The co-ordinates of P1 are 2 and 3
The co-ordinates of P2 are 5 and 5

5.4 Unions inside Structures

- ✓ Union can be very useful when declared inside a structure. Consider an example in which you want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario.

```
struct student
{
    union
    {
        char name[20];
        introll_no;
    };
    int marks;
};

void main()
{
    struct student stud;
    char choice;
    printf("\n You can enter the name or roll number of the student");
    printf("\n Do you want to enter the name? (Yes or No) : ");
    scanf(&choice);
    if(choice=='y' || choice=='Y')
    {
        printf("\n Enter the name : ");
        scanf("%s",&stud.name);
    }
    else
    {   printf("\n Enter the roll number : ");
        scanf("%d", &stud.roll_no);
    }
    printf("\n Enter the marks : ");
    scanf("%d", &stud.marks);
    if(choice=='y' || choice=='Y')
        printf("\n Name : %s ", stud.name);
    else
        printf("\n Roll Number : %d ", stud.roll_no);
        printf("\n Marks : %d", stud.marks);
```

}

5.5 Enumerated Data Types

- ✓ The enumerated data type is a **user defined type** based on the **standard integer type.**
- ✓ An enumeration consists of a set of named integer constants. That is, in an enumerated type, **each integer value is assigned an identifier**. This identifier (also known as an enumeration constant) can be used as **symbolic names to make the program more readable.**
- ✓ To define enumerated data types, **enum** keyword is used.
- ✓ Enumerations create new data types to contain values that are not limited to the values fundamental data types may take. The syntax of creating an enumerated data type can be given as below.

  enum enumeration_name{ identifier1, identifier2, ….., identifier n };
- ✓ Consider the example given below which creates a new type of variable called COLORS to store colors constants.

  enum COLORS {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE};
- ✓ In case you do not assign any value to a constant, the default value for the first one in the list - RED (in our case), has the value of 0. The rest of the undefined constants have a value 1 more than its previous one. So in our example,

  RED = 0, BLUE = 1, BLACK = 2, GREEN = 3, YELLOW = 4, PURPLE = 5, WHITE =6
- ✓ If you want to explicitly assign values to these integer constants, then you should specifically mention those values as shown below.

  enum COLORS {RED = 2, BLUE, BLACK = 5, GREEN = 7, YELLOW, PURPLE, WHITE = 15};
- ✓ As a result of the above statement, now

  RED = 2, BLUE = 3, BLACK = 5, GREEN = 7, YELLOW = 8, PURPLE = 9, WHITE = 15

5.5.1 Enum Variables

- ✓ The syntax for declaring a variable of an enumerated data type can be given as,

  enum enumeration_name variable_name;
- ✓ So to create a variable of COLORS, we may write:

  enum COLORS bg_color;
- ✓ Another way to declare a variable can be as illustrated in the statement below.

  enum COLORS {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}bg_color, fore_color;

5.5.2 Using the Typedef Keyword

- ✓ C permits to use typedef keyword for enumerated data types. For ex, if we write

  typedef enum COLORS color;
- ✓ Then, we can straight-away declare variables by writing

  color forecolor = RED;

5.5.3 Assigning values to Enumerated Variables

- ✓ Once the enumerated variable has been declared, **values can be stored in it.**
- ✓ However, an enumerated variable can hold only declared values for the type.
- ✓ For example, to assign the color black to the back ground color, we will write,

  bg_color = BLACK;
- ✓ Once an enumerated variable has been assigned a value, we can store its value in another variable of the same type as shown below.

```
enum COLORS bg_color, border_color;
bg_color = BLACK;
border_color = bg_color;
```

5.5.4 Enumeration Type Conversion

- ✓ Enumerated types can be **implicitly or explicitly cast.**
- ✓ For ex, the compiler can implicitly cast an enumerated type to an integer when required.
- ✓ However, when we implicitly cast an integer to an enumerated type, the compiler will either generate an error or warning message.
- ✓ To understand this, answer one question. If we write:

```
enum COLORS{RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE};

enum COLORS c;

c = BLACK + WHITE;
```

- ✓ Here, c is an enumerate data type variable. If we write, c = BLACK + WHITE, then logically, it should be 2 + 6 = 8; which is basically a value of type int. However, the left hand side of the assignment operator is of the type enumCOLORS. SO the statement would complain an error.
- ✓ To remove the error, you can do either of two things. First, declare c to be an int.
- ✓ Second, cast the right hand side in the following manner. :

```
c = enum COLORS(BLACK + WHITE);
```

5.5.5 Comparing Enumerated Types

- ✓ C also allows using comparison operators on enumerated data type. Look at the following statements which illustrate this concept.

```
bg_color = (enum COLORS)6;
if(bg_color == WHITE)
    fore_color = BLUE;
fore_color = BLACK;
if(bg_color == fore_color)
    printf("\n NOT VISIBLE");
```

- ✓ Since enumerated types are derived from integer type, they can be used in a switch-case statement.

```
enum {RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE}bg_color;
switch(bg_color)
{
    case RED:
    case BLUE:
    case GREEN:
                printf("\n It is a primary color");
                break;
    case default:
                printf("\n It is not a primary color");
                break;
}
```

5.5.6 Input/Output Operations on Enumerated Types

- ✓ Since enumerated types are derived types, they cannot be read or written using formatted I/O functions available in C. When we read or write an enumerated type, we read/write it as an integer. The compiler would implicitly do the type conversion as discussed earlier.

```
enum COLORS(RED, BLUE, BLACK, GREEN, YELLOW, PURPLE, WHITE};
enum COLORS c;
c = RED;
printf("\n Color = %d", c);
```

5.5.7 Differences between structure and union

Structure	Union
The struct keyword is used to define it.	The union keyword is used to define it.

Syntax:
struct struct–name
{
data_type var–name;
data_type var–name;
.....
};
	Syntax:
	union union-name
	{
	data_type var-name;
	data_type var-name;
	…….
	};

Example:
struct student
{
int r_no;
char name[20];
char course[20];
float fees;
};
	Example:
	union student
	{
	int r_no;
	char name[20];
	char course[20];
	float fees;
	};

A variable of structure student can be defined by writing: **struct student stud1;**	A variable of union student can be defined by writing: **union student stud1;**

Several members of a structure can be initialized at once.
Ex:
struct student stud1 = {01, "Rahul", "BCA", 45000};
	Individual members of a structure can be initialized one by one.
	Ex:
	union student stud1;
	stud1. r_no=01;
	stud1. name= "Rahul";
	stud1. course="BCA";
	stud1. fees =45000;

Each member within structure is assigned unique storage area of location.	The members or fields share the same memory space, so fresh data replaces any existing data.
The size of the structure is the sum of all members or fields.	The size of the union is the size of the largest member or field.
Altering the value of a member will not affect other members of the structure.	Altering any value of a member will alter other member values.

Module 5

Files

5.6 Introduction to Files

- ✓ **A file is a collection of data stored on a secondary storage device like hard disk.**
- ✓ A file is basically used because real life applications involve large amounts of data and in such situations the **console oriented I/O operations pose two major problems:**
  - First, it becomes **cumbersome and time consuming** to handle huge amount of data through terminals.
  - Second, when doing I/O using terminal, the **entire data is lost when either the program is terminated or computer is turned off**. Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

5.6.1 Streams in C

- ✓ In C, the standard streams are termed as **pre-connected input and output channels between a text terminal and the program (when it begins execution).**
- ✓ Therefore, **stream is a logical interface to the devices that are connected to the computer.**
- ✓ Stream is widely used as a logical interface to a file where a file can refer to a disk file, the computer screen, keyboard, etc.
- ✓ Although files may differ in the form and capabilities, all streams are the same.
- ✓ The three standard streams (figure 16.1) in C languages are- standard input (stdin), standard output (stdout) and standard error (stderr).
  - **Standard input (stdin):** Standard input is the **stream from which the program receives its data.** The program requests transfer of data using the **read operation.** However, not all programs require input. Generally, unless redirected, input for a program is expected from the keyboard.
  - **Standard output (stdout):** Standard output is **the stream where a program writes its output data.** The program requests data transfer using the **write operation.** However, not all programs generate output.
  - **Standard error (stderr):** Standard error is basically an **output stream used by programs to report error messages or diagnostics.** It is a stream independent of standard output and can be redirected separately. No doubt, the standard output and standard error can also be directed to the same destination.
- ✓ A stream is linked to a file using an open operation and disassociated from a file using a close operation.

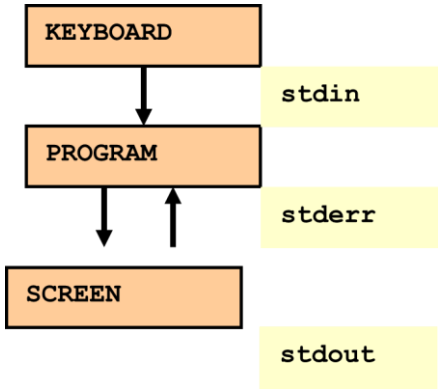


Figure 16.1 Standard Streams

5.6.2 Buffer Associated with File Stream

- ✓ When a **stream linked to a disk file is created**, a **buffer is automatically created and associated with the stream.**
- ✓ **A buffer is nothing but a block of memory that is used for temporary storage of data that has to be read from or written to a file.**
- ✓ Buffers are needed because **disk drives are block oriented devices** as they can operate efficiently when data has to be read/ written in blocks of certain size. The size of ideal buffer size is hardware dependant.
- ✓ **The buffer acts as an interface between the stream (which is character-oriented) and the disk hardware (which is block oriented).**
- ✓ When the program has to write data to the stream, it is saved in the buffer till it is full. Then the entire contents of the buffer are written to the disk as a block as shown in figure 16.2.

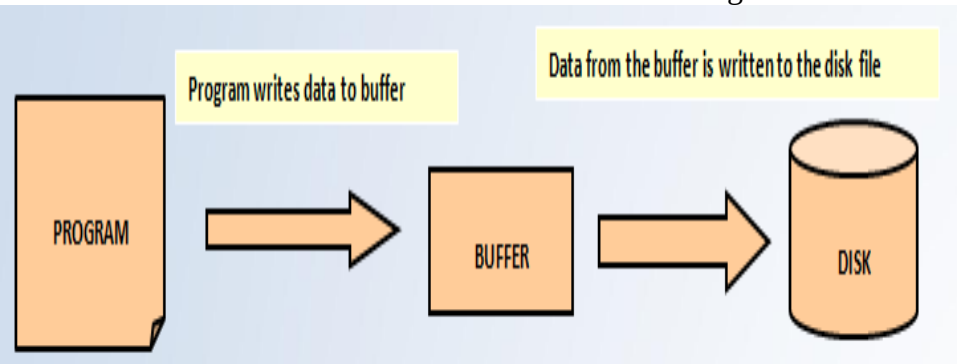


Figure 16.2 Buffers associated with streams

- ✓ Similarly, when reading data from a disk file, the data is read as a block from the file and written into the buffer. The program reads data from the buffer. **The creation and operation of the buffer is automatically handled by the operating system.** However, C provides some functions for buffer manipulation. The data resides in the buffer until the buffer is flushed or written to a file.

5.6.3 Types of Files

- ✓ In C, the types of files used can be broadly classified into two categories- **text files and binary files.**

1. ASCII Text files

- ✓ **A text file is a stream of characters that can be sequentially processed by a computer in forward direction.** For this reason a text file is usually opened for only **one kind of operation** (reading, writing, or appending) at any given time.
- ✓ Because text files only process characters, they can only **read or write data one character at a time.**
- ✓ In a text file, each line contains zero or more characters and ends with one or more characters that specify the end of line. **Each line** in a text file can have **maximum of 255 characters.**
- ✓ A **line** in a text file is not a C string, so it is **not terminated by a null character**.
- ✓ When data is written to a text file, each newline character is converted to a **carriage return/line feed character.** Similarly, when data is read from a text file, each carriage return/ line feed character is converted in to newline character.
- ✓ Another important thing is that when a text file is used, there are actually two representations of data- **internal or external.** For ex, an int value will be represented as 2 or 4 bytes of memory internally but externally the int value will be represented as a string of characters representing its decimal or hexadecimal value. To convert internal representation into external, we can use printf and fprintf functions. Similarly, to convert an external representation into internal scanf and fscanf can be used.

2. Binary Files

- ✓ **A binary file is a file which may contain any type of data, encoded in binary form for computer storage and processing purposes.**
- ✓ Like a text file, a **binary file is a collection of bytes.**
- ✓ Note that in C a byte and a character are equivalent. Therefore, a binary file is also referred to as a character stream with following two essential differences.
  - A binary file **does not require any special processing of the data** and each byte of data is transferred to or from the disk unprocessed.
  - C places **no constructs on the file**, and it may be **read from**, or **written to**, in **any manner the programmer wants.**
- ✓ Binary files store data in the **internal representation format**. Therefore, an int value will be stored I binary form as 2 or byte value. The same format is used to store data in memory as well as in file. Like text file, **binary file also ends with an EOF marker.**
- ✓ Binary files can be either processed **sequentially or randomly.**
- ✓ In a text file, an integer value 123 will be stored as a sequence of three characters- 1, 2 and 3. So each character will take 1 byte and therefore, to store the integer value 123 we need 3 bytes. However, in a binary file, the int value 123 will be stored in 2 bytes in the binary form. This clearly indicates that **binary files takes less space to store the same piece of data and eliminates conversion between internal and external representations and are thus more efficient than the text files.**

5.7 Using Files in C

- ✓ To use files in C, we must follow the steps given below.
  - **Declare a file pointer variable**
  - **Open the file**
  - **Process the file**
  - **Close the file**

1. Declaring a File Pointer Variable

- ✓ There can be a number of files on the disk. In order to access a particular file, you must **specify the name of the file that has to be used.**
- ✓ This is **accomplished by using a file pointer variable** that points to a structure FILE (defined in stdio.h).
- ✓ The file pointer will then be used in all subsequent operations in the file.
- ✓ The syntax for declaring a file pointer is

  FILE *file_pointer_name;
- ✓ For example, if we write

  FILE *fp;

  Then, fp is declared as a file pointer.
- ✓ An error will be generated if you use the filename to access a file rather than the file pointer

2. Opening a File

- ✓ **A file must be first opened before data can be read from it or written to it.**
- ✓ In order to **open a file** and associate it with a stream, the **fopen() function is used.**
- ✓ The prototype of fopen() can be given as:

  FILE *fopen(const char *file_name, const char *mode);
- ✓ Using the above prototype, the file whose pathname is the string pointed to by file_name is opened in the mode specified using the mode.
- ✓ **If successful, fopen() returns a pointer-to-structure and if it fails, it returns NULL.**

File Name

- ✓ Every file on the disk has a name associated with it.
- ✓ In DOS the file name can have one to eight characters optionally followed by a period and an extension that has one to three characters.
- ✓ Windows and UNIX permit filenames having maximum of 256 characters.
- ✓ In C, fopen() may contain the path information instead of specifying the filename. The path gives information about the location of the file on the disk.

File Mode

- ✓ Mode conveys to C the type of processing that will be done with the file.
- ✓ The different modes in which a file can be opened for processing are given in Table below:

MODE	DESCRIPTION
r	Open a text file for reading. If the stream (file) does not exist then an error will be reported.
w	Open a text file for writing. If the stream does not exist then it is created otherwise if the file already exists, then its contents would be deleted
a	Append to a text file. if the file does not exist, it is created.
rb	Open a binary file for reading. B indicates binary. By default this will be a sequential file in Media 4 format
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for both reading and writing. The stream will be positioned at the beginning of the file. When you specify "r+", you indicate that you want to read the file before you write to it. Thus the file must already exist.
w+	Open a text file for both reading and writing. The stream will be created if it does not exist, and will be truncated if it exists.
a+	Open a text file for both reading and writing. The stream will be positioned at the end of the file content.
r+b/ rb+	Open a binary file for read/write
w+b/wb+	Create a binary file for read/write
a+b/ab+	Append a binary file for read/write

- ✓ The fopen() can fail to open the specified file under certain conditions that are listed below:
  - Opening a file that is not ready for usage.
  - Opening a file that is specified to be on a non-existent directory/drive.
  - Opening a non-existent file for reading.
  - Opening a file to which access is not permitted.

Ex:

```
FILE *fp;
fp = fopen("Student.DAT", "r");
if(fp==NULL)
{
        printf("\n The file could not be opened");
        exit(1);
}
OR
char filename[30];
FILE *fp;
gets(filename);
fp = fopen(filename, "r+");
```

```
if(fp==NULL)
{
        printf("\n The file could not be opened");
        exit(1);
}
```

3. Closing a File Using fclose()

- ✓ To close an opened file, the fclose() function is used which **disconnects a file pointer from a file.**
- ✓ After the **fclose() has disconnected** the file pointer from the file, **the pointer can be used to access a different file or the same file but in a different mode.**
- ✓ The fclose() function not only closes the file but also **flushes all the buffers** that are maintained for that file
- ✓ If **you do not close a file** after using it, the **system closes it automatically** when the program exits. However, since there is a limit on the number of files which can be open simultaneously; the programmer must close a file when it has been used.
- ✓ The prototype of the fclose() function can be given as,

int fclose(FILE *fp);

Here, fp is a file pointer which points to the file that has to be closed. The function returns an integer value which indicates whether the fclose() was successful or not. A zero is returned if the function was successful; and a non-zero value is returned if an error occurred.

5.8 Read Data from Files

- ✓ C provides the following set of functions to read data from a file.
  - **fscanf()**
  - **fgets()**
  - **fgetc()**
  - **fread()**

1. fscanf()

- ✓ **The fscanf() is used to read formatted data from the stream.**
- ✓ The syntax of the fscanf() can be given as,

int fscanf(FILE *stream, const char *format,…);

- ✓ **The fscanf() is used to read data from the stream and store them according to the parameter format into the locations pointed by the additional arguments.**

Ex:

```
#include<stdio.h>
void main()
{
        FILE *fp;
        char name[80];
        int roll_no;
        fp = fopen("Student.DAT", "r");
        if(fp==NULL)
        {
            printf("\n The file could not be opened");
            exit(1);
        }
        printf("\n Enter the name and roll number of the student : ");
        fscanf(stdin, "%s %d", name, &roll_no);  /* read from keyboard */
```

```
        printf("\n NAME : %s \t ROLL NUMBER = %d", name, roll_no);  // READ FROM FILE-
                                                                    Student.DAT
        fscanf(fp, "%s %d", name, &roll_no);
        printf("\n NAME : %s \t ROLL NUMBER = %d", name, roll_no);
        fclose(fp);
}
```

2. fgets()

- ✓ fgets() stands for **file get string.**
- ✓ **The fgets() function is used to get a string from a stream.**
- ✓ The syntax of fgets() can be given as:

  char *fgets(char *str, int *size*, FILE *stream);
- ✓ **The fgets() function reads at most one less than the number of characters specified by size (gets size - 1 characters) from the given stream and stores them in the string str. The fgets() terminates as soon as it encounters either a newline character or end-of-file or any other error.** However, if a newline character is encountered it is retained. When all the characters are read without any error, a '\0' character is appended to end the string.

Ex:

```
#include<stdio.h>
void main()
{
        FILE *fp;
        char str[80];
        fp = fopen("Student.DAT", "r");
        if(fp==NULL)
        {
            printf("\n The file could not be opened");
            exit(1);
        }
        while (fgets(str, 80, fp) != NULL)
            printf("\n %s", str);
        printf("\n\n File Read. Now closing the file");
        fclose(fp);
}
```

3. fgetc()

- ✓ **The fgetc() function returns the next character from stream, or EOF if the end of file is reached or if there is an error.**
- ✓ The syntax of fgetc() can be given as

  int fgetc(FILE *stream);
- ✓ **fgetc returns the character read as an int or return EOF to indicate an error or end of file.**
- ✓ fgetc() reads a single character from the current position of a file (file associated with *stream)*. After reading the character, the function increments the associated file pointer (if defined) to point to the next character. However, if the stream has already reached the end of file, the end-of-file indicator for the stream is set.

Ex: Write a C program to copy a text file to another, read both the input file name and target file name.

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

```
void main()
{
   FILE *fptr1, *fptr2;
   char filename1[100], filename2[100],ch;

   printf("Enter the filename to open for reading: \n");
   scanf("%s", filename1);

   fptr1 = fopen(filename1, "r");
   if (fptr1 == NULL)
   {
      printf("Error in opening the file");
      exit(0);
   }

   printf("Enter the filename to open for writing: \n");
   scanf("%s", filename2);

   fptr2 = fopen(filename2, "w");
   if (fptr2 == NULL)
   {
      printf("Error in opening the file");
      exit(0);
   }

   while (!feof(fptr1))
   {
       ch=fgetc(fptr1);
       fputc(ch,fptr2);
   }

   printf("\n Contents copied to %s", filename2);

   fclose(fptr1);
   fclose(fptr2);
}
```

4. fread()

- ✓ **The fread() function is used to read data from a file.**
- ✓ Its syntax can be given as

  int fread(void *str, size_t size, size_t num, FILE *stream);
- ✓ The function fread() reads num number of objects (where each object is size bytes) and places them into the array pointed to by str. The data is read from the given input stream.
- ✓ **Upon successful completion, fread() returns the number of bytes successfully read.** The number of objects will be less than num if a read error or end-of-file is encountered. If size or num is 0, fread()

will return 0 and the contents of str and the state of the stream remain unchanged. In case of error, the error indicator for the stream will be set.

- ✓ The fread() function advances the file position indicator for the stream by the number of bytes read.

Ex:

```
#include <stdio.h>
void main()
{
        FILE *fp;
        char str[10];
        fp = fopen("Letter.TXT", "r+");
        if(fp==NULL)
        {
             printf("\n The file could not be opened");
             exit(1);
        }
        fread(str, 1, 10, fp);
        str[10]= '\0';
        printf("\n First 9 characters of the file are : %s", str);
        fclose(fp);
 }
```

5.9 Writing Data to Files

- ✓ C provides the following set of functions to read data from a file.
  - **fprintf()**
  - **fputs()**
  - **fputc()**
  - **fwrite()**

1. fprintf()

- ✓ **The fpritnt() is used to write formatted output to stream.**
- ✓ Its syntax can be given as,

  int fprintf (FILE * stream, const char * format, ...);
- ✓ The function writes to the specified stream, data that is formatted as specified by the format argument. After the format parameter, the function can have as many additional arguments as specified in format.
- ✓ The parameter format in the fprintf() is nothing but a C string that contains the text that has to be written on to the stream.

Ex:

```
#include <stdio.h>
void main()
{
        FILE *fp;
        int i;
        char name[20];
        float salary;
        fp = fopen("Details.TXT", "w");
        if(fp==NULL)
        {
             printf("\n The file could not be opened");
```

```
            exit(1);
        }
        for(i=0;i<10;i++)
        {
            printf ("\n Enter your name : ");
            gets(name);
            printf ("\n Enter your salary : ");
            scanf("%f", &salary);
            fprintf(fp, " NAME : %s \t SALARY: %f", name, salary);
    }
    fclose(fp);
}
```

2. fputs()

- ✓ **The fputs() is used to write a line into a file.**
- ✓ The syntax of fputs() can be given as

  int fputs(const char *str, FILE *stream);
- ✓ The fputs() writes the string pointed to by str to the stream pointed to by stream.
- ✓ **On successful completion, fputs() returns 0. In case of any error, fputs() returns EOF.**

Ex:

```
#include<stdio.h>
void main()
{
        FILE *fp;
        char feedback[100];
        fp = fopen("Comments.TXT", "w");
        if(fp==NULL)
        {
            printf("\n The file could not be opened");
            exit(1);
        }
        printf("\n Kindly give the feedback on this book : ");
        gets(feedback);
        fputs(feedback, fp);
        fclose(fp);
}
```

3. fputc()

- ✓ **The fputc() is used to write a character to the stream.**
- ✓ The syntax of fputc() can be given as,

  int fputc(int c, FILE *stream);
- ✓ **The fputc() function will write the byte specified by c (converted to an unsigned char) to the output stream pointed to by stream.** Upon successful completion, fputc() will return the value it has written. Otherwise, in case of error, the function will return EOF and the error indicator for the stream will be set.

Ex:

```
#include<stdio.h>
void main()
{
        FILE *fp;
        char feedback[100];
```

```
        int i;
        fp = fopen("Comments.TXT", "w");
        if(fp==NULL)
        {
            printf("\n The file could not be opened");
            exit(1);
        }
        printf("\n Kindly give the feedback on this book : ");
        gets(feedback);
        for(i=0;i<feedback[i];i++)
            fputc(feedback[i], fp);
        fclose(fp);
}
```

4. fwrite()

- ✓ **The fwrite() is used to write data to a file.**
- ✓ The syntax of fwrite() can be given as,

  int fwrite(const void *str, size_t size, size_t count, FILE *stream);
- ✓ The fwrite() function will write, from the array pointed to by str, up to count objects of size specified by size, to the stream pointed to by stream.
- ✓ The file-position indicator for the stream (if defined) will be advanced by the number of bytes successfully written. In case of error, the error indicator for the stream will be set.

Ex:

```
#include<stdio.h>
void main()
{
        FILE *fp;
        size_t count;
        char str[] = "GOOD MORNING ";
        fp = fopen("Welcome.txt", "wb");
        if(fp==NULL)
        {
            printf("\n The file could not be opened");
            exit(1);
        }
        count = fwrite(str, 1, strlen(str), fp);
        printf("\n %d bytes were written to the files", count);
        fclose(fp);
}
```

- ✓ fwrite() can be used to write characters, integers, structures, etc to a file. However, fwrite() can be used only with files that are opened in binary mode.

5.10 Detecting the End-Of-File

- ✓ In C, there are two ways to detect the end-of-file.

1. While reading the file in text mode, character by character, the programmer can compare the character that has been read with the EOF, which is a symbolic constant defined in stdio.h with a value -1.

```
while(1)
{
        c = fgetc(fp);        // here c is an int variable
```

```
        if (c==EOF)
            break;
        printf("%c", c);
    }
```

2. The other way is to use the standard library function feof() which is defined in stdio.h. The feof() is used to distinguish between two cases

- **When a stream operation has reached the end of a file**
- **When the EOF ("end of file") error code has been returned as a generic error indicator even when the end of the file has not been reached**

✓ The prototype of feof() can be given as:

int feof(FILE *fp);

✓ feof() returns zero (false) when the end of file has not been reached and a one (true) if the end-of-file has been reached.

```
while( !feof(fp)
{   fgets(str, 80, fp);
    printf("\n %s", str);
}
```