CHAPTER 1: Introduction to computer hardware and software

1 Computer Generations:

Computers are such an integral part of our everyday life now most people take them and what they have added to life totally for granted. Even more so the generation who have grown from infancy within the global desktop and laptop revolution since the 1980s. The history of the computer goes back several decades however and there are five definable generations of computers.

Each generation is defined by a significant technological development that changes fundamentally how computers operate – leading to more compact, less expensive, but more powerful, efficient and robust machines.

1940 – 1956: First Generation – Vacuum Tubes

These early computers used vacuum tubes as circuitry and magnetic drums for memory. As a result they were enormous, literally taking up entire rooms and costing a fortune to run. These were inefficient materials which generated a lot of heat, sucked huge electricity and subsequently generated a lot of heat which caused ongoing breakdowns.

These first generation computers relied on ‘machine language’ (which is the most basic programming language that can be understood by computers). These computers were limited to solving one problem at a time. Input was based on punched cards and paper tape. Output came out on print-outs. The two notable machines of this era were the UNIVAC and ENIAC machines – the UNIVAC is the first every commercial computer which was purchased in 1951 by a business – the US Census Bureau.

The replacement of vacuum tubes by transistors saw the advent of the second generation of computing. Although first invented in 1947, transistors weren’t used significantly in computers until the end of the 1950s. They were a big improvement over the vacuum tube, despite still subjecting computers to damaging levels of heat. However they were hugely superior to the vacuum tubes, making computers smaller, faster, cheaper and less heavy on electricity use. They still relied on punched card for input/printouts.

The language evolved from cryptic binary language to symbolic (‘assembly’) languages. This meant programmers could create instructions in words. About the same time high level programming languages were being developed (early versions of COBOL and FORTRAN). Transistor-driven machines were the first computers to store instructions into their memories – moving from magnetic drum to magnetic core ‘technology’. The early versions of these machines were developed for the atomic energy industry.
Fig. 1.2: Transistors


By this phase, transistors were now being miniaturised and put on silicon chips (called semiconductors). This led to a massive increase in speed and efficiency of these machines. These were the first computers where users interacted using keyboards and monitors which interfaced with an operating system, a significant leap up from the punch cards and printouts. This enabled these machines to run several applications at once using a central program which functioned to monitor memory.

As a result of these advances which again made machines cheaper and smaller, a new mass market of users emerged during the ‘60s.

Fig. 1.3: Third generation Computers


This revolution can be summed in one word: Intel. The chip-maker developed the Intel 4004 chip in 1971, which positioned all computer components (CPU, memory, input/output controls) onto a single chip. What filled a room in the 1940s now fit in the palm of the hand. The Intel chip housed thousands of integrated circuits. The year 1981 saw the first ever computer (IBM) specifically designed for home use and 1984 saw the MacIntosh introduced by Apple. Microprocessors even moved beyond the realm of computers and into an increasing number of everyday products.
The increased power of these small computers meant they could be linked, creating networks. Which ultimately led to the development, birth and rapid evolution of the Internet. Other major advances during this period have been the Graphical user interface (GUI), the mouse and more recently the astounding advances in lap-top capability and hand-held devices.

Fig. 1.4: Microprocessors

2010-: Fifth Generation – Artificial Intelligence

Computer devices with artificial intelligence are still in development, but some of these technologies are beginning to emerge and be used such as voice recognition. AI is a reality made possible by using parallel processing and superconductors. Leaning to the future, computers will be radically transformed again by quantum computation, molecular and nano technology.

The essence of fifth generation will be using these technologies to ultimately create machines which can process and respond to natural language, and have capability to learn and organize themselves.
Fig. 1.5: Artificial Intelligence

2 Computer Types:-
Since the advent of the first computer different types and sizes of computers are offering different services. Computers can be as big as occupying a large building and as small as a laptop or a microcontroller in mobile & embedded systems.

The four basic types of computers are as under:
1. Supercomputer
2. Mainframe Computer
3. Minicomputer
4. Microcomputer

- Supercomputer
  The most powerful computers in terms of performance and data processing are the Supercomputers. These are specialized and task specific computers used by large organizations. These computers are used for research and exploration purposes, like NASA uses supercomputers for launching space shuttles, controlling them and for space exploration purposes.

  The supercomputers are very expensive and very large in size. It can be accommodated in large air-conditioned rooms; some super computers can span an entire building.

Fig. 1.6: Supercomputers
• Mainframe Computer

Although Mainframes are not as powerful as supercomputers, but certainly they are quite expensive nonetheless, and many large firms & government organizations use Mainframes to run their business operations. The Mainframe computers can be accommodated in large air-conditioned rooms because of its size. Super-computers are the fastest computers with large data storage capacity, Mainframes can also process & store large amount of data. Banks educational institutions & insurance companies use mainframe computers to store data about their customers, students & insurance policy holders.

Fig. 1.7: Mainframe Computers

• Mini Computer

Minicomputers are used by small businesses & firms. Minicomputers are also called as “Midrange Computers”. These are small machines and can be accommodated on a disk with not as processing and data storage capabilities as super-computers & Mainframes.

These computers are not designed for a single user. Individual departments of a large company or organizations use Mini-computers for specific purposes. For example, a production department can use Mini-computers for monitoring certain production process.
Micro Computer

Desktop computers, laptops, personal digital assistant (PDA), tablets & smartphones are all types of microcomputers. The micro-computers are widely used & the fastest growing computers. These computers are the cheapest among the other three types of computers. The Micro-computers are specially designed for general usage like entertainment, education and work purposes. Well known manufacturers of Micro-computer are Dell, Apple, Samsung, Sony& Toshiba.

Desktop computers, Gaming consoles, Sound & Navigation system of a car, Netbooks, Notebooks, PDA’s, Tablet PC’s, Smartphones, Calculators are all type of Microcomputers.
a. Bits, bytes and words

A bit is a **Binary digit**. So a bit is a zero or a one. Bits can be implemented in computer hardware using switches. If the switch is on then the bit is one and if the switch is off then the bit is zero. A bit is limited to representing two values.

Since the alphabet contains more than two letters, a letter cannot be represented by a bit. A byte is a sequence of bits. Since the mid 1960's a byte has been 8 bits in length. 01000001 is an example of a byte. Since there are 8 bits in a byte there are $2^8$ different possible sequences for one byte, ranging from 00000000 to 11111111. This means that a byte can be used to represent any type of value with no more than $2^8 = 256$ possible values. Since the number of things that you can enter on a computer keyboard is smaller than 256 (including all keystone pairs, like shift or control plus another key), a code for a keystone is represented with a code within a byte.

Since characters (letters, decimal digits and special characters such as punctuation marks, etc) can be represented with bytes, a standard is needed to ensure that the code that's used on your computer is the same as the code that is used on mine. There are two standard codes that use one byte to represent a character, ASCII and EBCDIC. ASCII, the American Standard Code for Information Interchange, is the code that is most commonly used today. EBCDIC, Extended Binary Coded Decimal Interchange Code, was used by IBM on its large mainframe computers in the past. Wikipedia has more than you want to know about ASCII and EBCDIC. Since these codes are limited to 256 possible combinations, certain character sets, such as Chinese, Arabic, Japanese, Klingon and others, cannot be represented using these codes. This problem is solved by using another code, Unicode, which uses 2 bytes for each character. This extension allows $2^{16}$ different symbols to be represented, a total of 65,536. The use of Unicode gives more flexibility in the representation of data. The drawback of using Unicode is that it takes twice as much space to store the same number of characters.

A word is the number of bits that are manipulated as a unit by the particular CPU of the computer. Today most CPUs have a word size of 32 or 64 bits. For example, the notebook computer that I bought in May 2008 contains a core 2 duo 64 bit processor. Data is fetched from memory to the processor in word size chunks and manipulated by the ALU in word size chunks.
All other things being equal, (and they never are), larger word size implies faster and more flexible processing.

b. CPU

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions. The computer industry has used the term "central processing unit" at least since the early 1960s.[1] Traditionally, the term "CPU" refers to a processor, more specifically to its processing unit and control unit (CU), distinguishing these core elements of a computer from external components such as main memory and I/O circuitry.[2]

The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains almost unchanged. Principal components of a CPU include the arithmetic logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations and a control unit that orchestrates the fetching (from memory) and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

This part of the computer system collects the raw data from the input devices and converts it to useful information which can then be used by the output devices. On some computers, the CPU can be a single microchip. On bigger systems, the CPU can be formed from a number of chips working together.

The CPU is made up of three main parts:

- Control Unit this part controls the input and output devices

- Arithmetic Logic Unit this is the part that does all the working out: it does all the maths and makes the decisions

- Immediate Access Store this is the memory available for programs and data. The more memory the CPU has - the more programs it can run at the same time; and the more data it can deal with in one go.
Fig. 1.10: CPU

c. Primary Memory

Primary storage (also known as main memory or internal memory), often referred to simply as memory, is the only one directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them as required. Any data actively operated on is also stored there in uniform manner.

Historically, early computers used delay lines, Williams tubes, or rotating magnetic drums as primary storage. By 1954, those unreliable methods were mostly replaced by magnetic core memory. Core memory remained dominant until the 1970s, when advances in integrated circuit technology allowed semiconductor memory to become economically competitive.

This led to modern random-access memory (RAM). It is small-sized, light, but quite expensive at the same time. (The particular types of RAM used for primary storage are also volatile, i.e. they lose the information when not powered).

Main memory is directly or indirectly connected to the central processing unit via a memory bus. It is actually two buses (not on the diagram): an address bus and a data bus. The CPU firstly sends a number through an address bus, a number called memory address, that indicates the desired location of data. Then it reads or writes the data in the memory cells using the data bus. Additionally, a memory management unit (MMU) is a small device between CPU
and RAM recalculating the actual memory address, for example to provide an abstraction of virtual memory or other tasks.

As the RAM types used for primary storage are volatile (uninitialized at start up), a computer containing only such storage would not have a source to read instructions from, in order to start the computer. Hence, non-volatile primary storage containing a small startup program (BIOS) is used to bootstrap the computer, that is, to read a larger program from non-volatile secondary storage to RAM and start to execute it.

![Primary Memory (Main Memory)](image)

**Fig. 1.11: Primary Memory**

d. **Secondary Memory**

Secondary storage (also known as external memory or auxiliary storage), differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers the desired data using intermediate area in primary storage. Secondary storage does not lose the data when the device is powered down—it is non-volatile. Per unit, it is typically also two orders of magnitude less expensive than primary storage. Modern computer systems typically have two orders of magnitude more secondary storage than primary storage and data are kept for a longer time there.

In modern computers, hard disk drives are usually used as secondary storage. The time taken to access a given byte of information stored on a hard disk is typically a few thousandths of a second, or milliseconds. By contrast, the time taken to access a given byte of information
stored in random-access memory is measured in billionths of a second, or nanoseconds. This illustrates the significant access-time difference which distinguishes solid-state memory from rotating magnetic storage devices: hard disks are typically about a million times slower than memory. Rotating optical storage devices, such as CD and DVD drives, have even longer access times. With disk drives, once the disk read/write head reaches the proper placement and the data of interest rotates under it, subsequent data on the track are very fast to access. To reduce the seek time and rotational latency, data are transferred to and from disks in large contiguous blocks.

Most computer operating systems use the concept of virtual memory, allowing utilization of more primary storage capacity than is physically available in the system. As the primary memory fills up, the system moves the least-used chunks (pages) to secondary storage devices (to a swap file or page file), retrieving them later when they are needed. As more of these retrievals from slower secondary storage are necessary, the more the overall system performance is degraded.

![Secondary Memory](image)

**Fig. 1.12: Secondary Memory**

### c. Ports and Connections

A Computer Port is an interface or a point of connection between the computer and its peripheral devices. Some of the common peripherals are mouse, keyboard, monitor or display unit, printer, speaker, flash drive etc.

The main function of a computer port is to act as a point of attachment, where the cable from the peripheral can be plugged in and allows data to flow from and to the device.
A computer port is also called as a Communication Port as it is responsible for communication between the computer and its peripheral device. Generally, the female end of the connector is referred to as a port and it usually sits on the motherboard.

In Computers, communication ports can be divided into two types based on the type or protocol used for communication. They are Serial Ports and Parallel Ports.

A serial port is an interface through which peripherals can be connected using a serial protocol which involves the transmission of data one bit at a time over a single communication line. The most common type of serial port is a D-Subminiature or a D-sub connector that carry RS-232 signals.

A parallel port, on the other hand, is an interface through which the communication between a computer and its peripheral device is in a parallel manner i.e. data is transferred in or out in parallel using more than one communication line or wire. Printer port is an example of parallel port.

**PS/2**

PS/2 connector is developed by IBM for connecting mouse and keyboard. It was introduced with IBM’s Personal Systems/2 series of computers and hence the name PS/2 connector. PS/2 connectors are color coded as purple for keyboard and green for mouse.

**Serial Port**

Even though the communication in PS/2 and USB is serial, technically, the term Serial Port is used to refer the interface that is compliant to RS-232 standard.

**Parallel Port**

Parallel port is an interface between computer and peripheral devices like printers with parallel communication. The Centronics port is a 36 pin port that was developed as an interface for printers and scanners and hence a parallel port is also called as a Centronics port.

Before the wide use of USB ports, parallel ports are very common in printers. The Centronics port was later replaced by DB-25 port with parallel interface.
Audio Ports

Audio ports are used to connect speakers or other audio output devices with the computer. The audio signals can be either analogue or digital and depending on that the port and its corresponding connector differ.

VGA Port

VGA port is found in many computers, projectors, video cards and High Definition TVs. It is a D-sub connector consisting of 15 pins in 3 rows. The connector is called as DE-15. VGA port is the main interface between computers and older CRT monitors. Even the modern LCD and LED monitors support VGA ports but the picture quality is reduced. VGA carries analogue video signals up to a resolution of 640X480.

Digital Video Interface (DVI)

DVI is a high speed digital interface between a display controller like a computer and a display device like a monitor. It was developed with an aim of transmitting lossless digital video signals and replace the analogue VGA technology.

Display Port

Display Port is a digital display interface with optional multiple channel audio and other forms of data. Display Port is developed with an aim of replacing VGA and DVI ports as the main interface between a computer and monitor.

HDMI

HDMI is an abbreviation of High Definition Media Interface. HDMI is a digital interface to connect High Definition and Ultra High Definition devices like Computer monitors, HDTVs, Blu-Ray players, gaming consoles, High Definition Cameras etc. HDMI can be used to carry uncompressed video and compressed or uncompressed audio signals.

USB

Universal Serial Bus (USB) replaced serial ports, parallel ports, PS/2 connectors, game ports and power chargers for portable devices.
USB port can be used to transfer data, act as an interface for peripherals and even act as power supply for devices connected to it. There are three kinds of USB ports: Type A, Type B or mini USB and Micro USB.

f. Input Devices
An input device is any hardware device that sends data to a computer, allowing you to interact with and control it. The most commonly used or primary input devices on a computer are the keyboard and mouse. However, there are dozens of other devices that can also be used to input data into the computer.

Today, input devices are important because they are what allows you to interact with and add new information to a computer. For example, if a computer had no input devices, it could run by itself but there would be no way to change its settings, fix errors, or other various user interactions. Also, if you wanted to add new information to the computer (e.g., text, command, document, picture, etc.), you wouldn't be able to do so without an input device.

Fig. 1.13: Input Devices

g. Output Devices
An output device is any device used to send data from a computer to another device or user. Most computer data output that is meant for humans is in the form of audio or video. Thus, most output devices used by humans are in these categories. Examples include monitors, projectors, speakers, headphones and printers.
Output devices allow computers to communicate with users and with other devices. This can include peripherals, which may be used for input/output (I/O) purposes, like network interface cards (NICs), modems, IR ports, RFID systems and wireless networking devices, as well as mechanical output devices, like solenoids, motors and other electromechanical devices.

![Output Devices](image)

**Fig. 1.14: Output Devices**

Some of the most common output devices that people are familiar with include monitors, which produce video output; speakers, which produce audio output; and printers, which produce text or graphical output.

**h. Computers in a network**

The **Network** allows computers to **connect and communicate** with different computers via any medium. LAN, MAN and WAN are the three major types of the network designed to operate over the area they cover. There are some similarities and dissimilarities between them. One of the major differences is the geographical area they cover, i.e. LAN covers the smallest area; MAN covers an area larger than LAN and WAN comprises the largest of all.

**Local Area Network (LAN):** LAN or Local Area Network connects network devices in such a way that personal computer and workstations can share data, tools and programs. The group of computers and devices are connected together by a switch, or stack of switches, using a private addressing scheme as defined by the TCP/IP protocol. Private addresses are unique in relation to
other computers on the local network. Routers are found at the boundary of a LAN, connecting them to the larger WAN.

![LAN](image_url)

**Fig. 1.15: Local Area Network**

**Metropolitan Area Network (MAN):** MAN or Metropolitan area Network covers a larger area than that of a LAN and smaller area as compared to WAN. It connects two or more computers that are apart but resides in the same or different cities. It covers a large geographical area and may serve as an ISP (Internet Service Provider). MAN is designed for customers who need a high-speed connectivity. Speeds of MAN ranges in terms of Mbps. It’s hard to design and maintain a Metropolitan Area Network.

The fault tolerance of a MAN is less and also there is more congestion in the network. It is costly and may or may not be owned by a single organization. The data transfer rate and the propagation delay of MAN is moderate. Devices used for transmission of data through MAN are: Modem and Wire/Cable. Examples of a MAN are the part of the telephone company network that can provide a high-speed DSL line to the customer or the cable TV network in a city.
**Wide Area Network (WAN):** WAN or Wide Area Network is a computer network that extends over a large geographical area, although it might be confined within the bounds of a state or country. A WAN could be a connection of LAN connecting to other LAN’s via telephone lines and radio waves and may be limited to an enterprise (a corporation or an organization) or accessible to the public. The technology is high speed and relatively expensive.

There are two types of WAN: Switched WAN and Point-to-Point WAN. WAN is difficult to design and maintain. Similar to a MAN, the fault tolerance of a WAN is less and there is more congestion in the network. A Communication medium used for WAN is PSTN or Satellite Link. Due to long distance transmission, the noise and error tend to be more in WAN.
i. Network hardware

Networking hardware, also known as network equipment or computer networking devices, are physical devices which are required for communication and interaction between devices on a computer network. Specifically, they mediate data in a computer network. Units which are the last receiver or generate data are called hosts or data terminal equipment.

Networking devices may include gateways, routers, network bridges, modems, wireless access points, networking cables, line drivers, switches, hubs, and repeaters; and may also include hybrid network devices such as multilayer switches, protocol converters, bridge routers, proxy servers, firewalls, network address translators, multiplexers, network interface controllers, wireless network interface controllers, ISDN terminal adapters and other related hardware.

The most common kind of networking hardware today is a copper-based Ethernet adapter which is a standard inclusion on most modern computer systems. Wireless networking has become increasingly popular, especially for portable and handheld devices.

Other networking hardware used in computers includes data center equipment (such as file servers, database servers and storage areas), network services (such as DNS, DHCP, email, etc.) as well as devices which assure content delivery.

Taking a wider view, mobile phones, PDAs and even modern coffee machines may also be considered networking hardware. As technology advances and IP-based networks are integrated into building infrastructure and household utilities, network hardware will become an ambiguous term owing to the vastly increasing number of "network capable" endpoints.

Typical core network devices include:

- **Gateway**: an interface providing a compatibility between networks by converting transmission speeds, protocols, codes, or security measures.\(^2\)
- **Router**: a networking device that forwards data packets between computer networks. Routers perform the "traffic directing" functions on the Internet. A data packet is typically forwarded from one router to another through the networks that constitute the internetwork until it reaches its destination node.\[^3\] It works on OSI layer 3.\[^4\]

- **Switch**: a device that connects devices together on a computer network, by using packet switching to receive, process and forward data to the destination device. Unlike less advanced network hubs, a network switch forwards data only to one or multiple devices that need to receive it, rather than broadcasting the same data out of each of its ports.\[^5\] It works on OSI layer 2.

- **Bridge**: a device that connects multiple network segments. It works on OSI layers 1 and 2.\[^6\]

- **Repeater**: an electronic device that receives a signal and retransmits it at a higher level or higher power, or onto the other side of an obstruction, so that the signal can cover longer distances.\[^7\]

- **Repeater hub**: for connecting multiple Ethernet devices together and making them act as a single network segment. It has multiple input/output (I/O) ports, in which a signal introduced at the input of any port appears at the output of every port except the original incoming.\[^1\] A hub works at the physical layer (layer 1) of the OSI model.\[^8\] Repeater hubs also participate in collision detection, forwarding a jam signal to all ports if it detects a collision. Hubs are now largely obsolete, having been replaced by network switches except in very old installations or specialized applications.

![Network Diagram](image-url)\[Fig. 1.18: Networking Hardware\]
j. **Software basics and types**

A computer uses software, colloquially known as programs and applications, to perform tasks. The computer accesses its commands from the operating instructions in the software and then performs the tasks specified in these instructions. Computer users can purchase a host of software, including word processors, graphic editors, databases, games, and more. Software has to be compatible with a computer operating system, such as Windows, Mac OS, and Linux. A computer becomes unusable if it does not have software installed onto it. An operating system typically has software programs already installed onto it. In addition, numerous software programs can be found online totally free of charge.

A personal computer (PC) can eliminate the clutter sitting around the house. A computer has an electronic phone book, calendar, notepad, address book, daily planner, files, and folders. The latter two replaces manilla folders and filing cabinets, huge space wasters in any environment. People can use their computers to watch television and DVDs. Computers can play recorded music and streaming live audio. Computer technology has rapidly advanced to the point where people can upload entire photo albums, send electronic mail, and make long distance calls at the same time. More importantly, computer technology has increased its user-friendliness, making it easily accessible for those limited by mental and physical constraints.
3 Software Types

System Software

Software required to run the hardware parts of the computer and other application software are called system software. System software acts as interface between hardware and user applications. An interface is needed because hardware devices or machines and humans speak in different languages.

Based on its function, system software is of four types:

- Operating System
- Language Processor
- Device Drivers
**Operating System:** System software that is responsible for functioning of all hardware parts and their interoperability to carry out tasks successfully is called operating system (OS). OS is the first software to be loaded into computer memory when the computer is switched on and this is called booting. OS manages a computer’s basic functions like storing data in memory, retrieving files from storage devices, scheduling tasks based on priority, etc.

**Language Processor:** An important function of system software is to convert all user instructions into machine understandable language. When we talk of human machine interactions, languages are of three types:

- Machine-level language
- Assembly-level language
- High level language

System software that converts source code to object code is called language processor. There are three types of language interpreters:

- **Assembler** – Converts assembly level program into machine level program.
- **Interpreter** – Converts high level programs into machine level program line by line.
- **Compiler** – Converts high level programs into machine level programs at one go rather than line by line.

**Device Drivers:** System software that controls and monitors functioning of a specific device on computer is called device driver. Each device like printer, scanner, microphone, speaker, etc. that needs to be attached externally to the system has a specific driver associated with it. When you attach a new device, you need to install its driver so that the OS knows how it needs to be managed.

**Application Software**

A software that performs a single task and nothing else is called application software. Application software are very specialized in their function and approach to solving a problem.
So a spreadsheet software can only do operations with numbers and nothing else. A hospital management software will manage hospital activities and nothing else. Here are some commonly used application software –

- Word processing
- Spreadsheet
- Presentation
- Database management
- Multimedia tools

Utility Software

Application software that assist system software in doing their work is called utility software. Thus utility software is actually a cross between system software and application software. Examples of utility software include:

- Antivirus software
- Disk management tools
- File management tools
- Compression tools
- Backup tools

CHAPTER 2: Overview of C

HISTORY OF C

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure.

ALGOL gave the concept of structured programming to the computer science community.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) preliminary for writing system software. In 1970, Ken Thomson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. C was evolved from ALGOL, BCPL and B by Dennis Richie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with
the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating system in use today and the heart of the internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as “traditional C”. The language became more popular after publication of the book ‘The C programming Language’ by Brain Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as “K&R C” among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

American National Standards Institute (ANSI) Committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. The version of C is also referred to as C89. During 1990’s C++, a language entirely based on C, Underwent a number of improvements and changes and became a ANSI/ISO approved language in November 1977.

All popular computer language are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. The version is usually referred to as C99.

**IMPORTANCE OF C**

The increasing popularity of C is probably due to its many desirable qualities.

1. It is robust language whose rich set of built-in functions and operators used to write any complex program.
2. Programs written in C are efficient and fast.

For example, a program to increment a
variable from 0 to 15000 takes about one second in C more than 50 seconds in an interpreter BASIC.

These are only 32 keywords in ANSI C and its strength lies in its built-in functions.

3. C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

4. C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. This modular structure makes program debugging, testing and manufacturing easier.

5. Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously added our own functions to C library.

SAMPLE PROGRAM 1: PRINTING A MESSAGE

Consider a very simple program given in Fig 1.2.

Program

```c
main()
{
    /* printing begins */
    printf("I see, I remember");
    /* printing ends */
}
```

Fig 1.2 A Program to print one line of text

This program when executed will produce the following output:

    I see, I remember

The first line informs the system that the name of the program is main and the execution begins at this line. The main() is a special function used by the C system to tell the computer where the program starts. Every program must have exactly one main function. If we use more than one main function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following main indicates that the function main has no arguments (parameters).
The operation brace "{" in the second line marks the beginning of the function main and the closing brace "}" in the last line indicates the end of the line function. The function body contains three statements out of which only the printf line is an executable statement. The lines beginning with /* and ending with */ are known as comment lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between /* and */ is ignored by the compiler.

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once compiler finds an operating token, it ignores everything until it finds a closing token. The comment line

    /* ============ */

Is not valid and therefore results in an error.

Since comments do not affect the execution speed and size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program.

Let us now look at the printf() function, the only executable statement of the program.

    printf("I see, I remember");

Printf is a predefined standard C function for printing output. Predefined means that it is a function that has already been written and compiled, and linked together with our program at the time of linking.

The printf function causes everything between the starting and the ending quotation marks to be printed out. In case, the output will be:

    printf("I see, I remember");

Note that the printf line ends with a semicolon. Every statement in C should end with a semicolon(;) mark.

Suppose we want to print the above quotation in two lines as

    I see,
    I remember!

This can be achieved by adding another printf function as shown below:

    printf("I see, \n");
    printf("I see, I remember !");

The information contained between the parameters is called the arguments of the function. This argument of the first printf function is "I see, \n" and the second is " I remember !". These arguments are simply strings of characters to be printed out.

First printf contains a combination of two characters \ and \n at the end of the string.
The combination is collectively called the newline character. A newline character instructs the compiler to go to the next(new)line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma(,) the presence of the newline character \n causes the string “I remember!” to be printed on the next line. No space is allowed between \a and n.
If we omit the newline character from the first printf statement, then the output will again be a single line as shown below.

I see, I remember!

This is similar to the output of the program in the Fig 1.2. However, note that there is no space between, and I.
It is also possible to produce two or more lines of output by one printf statement with the use of newline character at appropriate places. For example, the statement

printf("I see, \n I remember!");

Will output

I see,
I remember!

While statement

printf(" I . . see, \n . . . . . . . . I . . . . . . . . . remember!");

Will print out

I

... see,

........ I

.......... remember!

#include<stdio.h>

At the beginning of all programs that use any input/output library functions. However, this is necessary for the functions printf and scanf which have been defined as a part of the C language.
C does make a distinction between uppercase and lowercase letters. For example, printf and PRINTF are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants.

The above example that printed I see, I remember is one of simplest programs. Figure 1.3 highlights the general format of such simple programs. All C programs need a main function.
The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- `main()`
- `int main()`
- `void main()`
- `main(void)`
- `void main(void)`
- `int main(void)`

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyboard `void` inside the parentheses. We may also specify the keyword `int` or `void` before the word `main`. The keyword `void` means that the function does not return any information to the operating system and `int` means that the function returns an integer value to the operating system. When `int` is specified, the last statement in the program must be “return 0”.

**SAMPLE PROGRAM 2: ADDING TWO NUMBERS**

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 1.4.

```c
/* Program ADDITION */
/* Written by EBG */
main()
{
    int number;
    float amount;
    /* line-1 */
    /* line-2 */
    /* line-3 */
    /* line-4 */
    /* line-5 */
    /* line-6 */
    /* line-7 */
```
number = 100;  /* line-8 */
/* line-9 */
amount = 30.75 + 75.35;  /* line-10 */
printf("\%d\n", number);  /* line-11 */
printf("\%5.2f", amount);  /* line-12 */
}
/* line-13 */

The first two lines of the program are comment lines. The words number and amount are variable names that are used to store numeric data. The numeric data may be either in integer from or in real form. In C, all variables should be declared to tell the compiler what are variable names are and what type of type od data they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

    int number;
    float amount;

Tell the compiler that number is an integer(int) ant amount is a floating(float) point number. Declaration statements must be appear at the beginning of the functions as shown in Fig.1.4. All declaration statements end with semicolon;
The words such as int and float are called the keywords and cannot be used as variable names.
Data is stored in a variable by assigning a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable number and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable amount, the statements

number = 100;
amount = 30.75-75.35;

are called the assignment statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of number. The print statement

printf("\%d\n", number);

Contains two arguments. The first argument "\%d" tells the compiler that the value of second argument number should be printed as decimal integer. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.

SAMPLE PROGRAM 3: INTEREST CALCULATION

    /*----------------------------- INVESTMENT PROBLEM ----------------------------*/
    #define PERIOD 10
```c
#define PRINCIPAL 5000.00

main()
{
    int year;
    float amount, value, inrate;

    amount = PRINCIPAL,
    inrate = 0.11;
    year = 0;

    while(year <= PERIOD)
    {
        printf(“%2d %8.2f
”, year, amount);
        value = amount + inrate * amount;
        year = year + 1;
        amount = value;
    }
}

SAMPLE PROGRAM 4: USE OF SUBROUTINES

main()
{
    int a, b, c;
    a = 5;
    b = 10;
    c = mul(a,b);

    printf(“multiplication of %d and %d is %d”, a, b, c);
}

mul(int a, int b);

MUL() FUNCTION STARTS
```
```c
int mul (int x, int y)
int p;

p = x*y;
{
    return(p);
}

/* -------------- MUL () FUNCTION ENDS ----------------*/
```

**BASIC STRUCTURE OF C PROGRAMS**

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 1.9.

The documentation section consists of a set of comment lines giving the name of the program, the author and the other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

![Figure 1.9 An overview of a C program](image-url)
There are some variables that are used in more than one function. Such variables are called global variables and are in the global declaration section that is outside of all the functions. This section also executable part. The declaration part declares all the variables used in the executable part. There is at one statement in the executable part. These two parts must appear between the opening and the closing braces. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon (;

**PROGRAMMING STYLE**

C is an free-form language. That is the C compiler does not care, where on the line we begin typing.

First of all, we must develop the First of all, we must develop the habit of writing programs in lowercase letters. **C** program statement are written in lowercase letters, uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of function. Indentation of braces and statements would make a program easier to read and debug.

Since C is a free-form language, we can group statements together on one line. The statements

```c
x = y + 1;

x = a + x;
```

can be written on one line as

```c
a = b; x = y + 1; z = a + x;
```

The program

```c
main()
{
    printf("Hello C");
}
```

may be written in one like

```c
main() { printf("Hello C");
```

However, this style make the program more difficult to understand and should not be used.

**EXECUTING A ‘C’ PROGRAM**

Executing a program written in C involves a series of steps. These are:

1. Creating the program
2. Compiling the program;
3. Linking the program with functions that are needed from the C library, and
4. Executing the program

Figure 1.10 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the operating system, system commands for implementing...
the
steps and conventions for naming files may differ on different systems.
The two most popular operating system today are UNIX and MS-DOS.

Figure 1.10 Process of compiling and running a C program

UNIX SYSTEM

Creating the Program
Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter c. Examples of valid file names are

hello.c
program.c
ebg1.c
The file is created with the help of a text editor, either ed or vi. The command for calling the editor and creating the file is

```
ed filename
```

if the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor.

When the editing is over, the file is saved on disk. It can then be referenced any time later by its name. The program that is entered into the file is known as the source program, since it represents the original form of the program.

**Compiling and Linking**
Let us assume that the source program has been created in a file named ebgl.c. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

```
cc ebgl.c
```

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name ebgl.o. This program is known as object code.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using exp() function, then the object code of this function should be brought from the main library of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the cc command is used.

If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again. The compiled and linked program is called the executable object code and is stored automatically in another file named a.out.
Executing the Program
Execution is a simple task. The command

```
a.out
```

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Something is wrong with the program logic or data. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

MS-DOS SYSTEM
The program can be created using any word processing software in non-document mode. The file name should end with the characters "c" like `program.c`, `pay.c`, etc. Then the command

```
MSC pay
```

under MS-DOS operating system would load the program stored in the file `pay.c` and generate the object code. This code is stored in another file under name `pay.obj`. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again. The linking is done by the command

```
LINK pay.obj
```

which generates the executable code with the filename `pay.exe`. Now the command `pay` would execute the program and give the results.

---

**CHAPTER 3: Constants, Variables, and Data types**

**CHARACTER SET**
The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces
The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

<table>
<thead>
<tr>
<th>Letters</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppercase A...Z</td>
<td>All decimal digits 0 ....9</td>
</tr>
<tr>
<td>Lowercase a...z</td>
<td></td>
</tr>
</tbody>
</table>

**Special Characters**
- comma
- period
- semicolon
- colon
- question mark
- apostrophe
- quotation mark
- exclamation mark
- vertical bar
- slash
- backslash
- tilde
- underscore
- $ dollar sign
- % percent sign
- & ampersand
- ^ caret
- * asterisk
- - minus sign
- + plus sign
- < opening angle bracket
- (or less than sign)
- > closing angle bracket
- (or greater than sign)
- ( left parenthesis
- ) right parenthesis
- [ left bracket
- ] right bracket
- { left brace
- } right brace
- # number sign

**C TOKENS**
In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens as shown in figure below. Programs are written using these tokens and the syntax of the language.

**KEYWORDS**
Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 1. All keywords must be written in lowercase.

---

**Table 2.1 C Character Set**

![Table 2.1 C Character Set](image)

**Fig 2. C Tokens and examples**

![Fig 2. C Tokens and examples](image)
Table 1 ANSI C Keywords

IDENTIFIERS
Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as first character. Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifier.

Rules for Identifiers
- First character must be an alphabet (or underscore).
- Must consist of only letters, digits or underscore.
- Only first 31 characters are significant.
- Cannot use a keyword.
- Must not contain white space.

CONSTANTS
Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in fig 3.

Fig 3. Basic types of constants

Integer Constants
An integer constant refers to a sequence of digits. There are three types of integers, namely,
- Decimal integer
- Octal integer
- Hexadecimal integer

**Decimal integers**

Decimal integers consist of a set of digits, 0-9, preceded by an optional – or + sign.

Examples:

123  -321 0 654321  +/8

Embedded spaces, commas, and non-digit characters are not permitted between digits.

Example: 15 750  20,000  $1000 are illegal numbers.

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integers are:

037  0 0435  0851

A sequence of digits preceded by 0x or 0X is considered as a hexadecimal integer. They may also include alphabets A through F or a through f. The letter A through F represents the numbers 10 through 15. Following are the examples of valid hex integers:

0X2  0x9f  0Xbcd  0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store large integer constants on these machines by appending qualifiers such as U, L and UL to the constants. Examples:

56789U or 56789u (unsigned integer)

987612347UL or 987612347ul (unsigned long integer)

9876543L or 9876543l (long integer)

**Program 2.1 : Representation of integer constants on a 16-bit computer**
Program

```c
main()
{
    printf("Integer values\n\n");
    printf("%d %d %d\n", 32767, 32767+1, 32767+10);
    printf("\n");
    printf("Long integer values\n\n");
    printf("%ld %ld %ld\n", 32767L, 32767L+1L, 32767L+10L);
}
```

Output

Integer values
32767 -32768 -32759
Long integer values
32767L 32768L 32777L

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real (or floating point) constants. Further examples of real constants are:

0.0083  -0.75  435.36  +247.0

A real number may also be expressed in exponential (or scientific) notation. For example, the value 215.65 may be written as \( 2.1565 \times 10^2 \) in exponential notation. \( E2 \) means multiply by \( 10^2 \). The general form is:

\[
\text{mantissa} \times 10^\text{exponent}
\]

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter \( e \) separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to “float”, this notation is said to represent a real number in floating point form. Examples of legal floating-point constants are:

0.65e1  12e2  15e+5  3.18E3  1.2E1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to -3.68E-7.
Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in the Table 4

### Table 4: Examples of Numeric Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Valid?</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>698354L</td>
<td>Yes</td>
<td>Represents long integer</td>
</tr>
<tr>
<td>25,000</td>
<td>No</td>
<td>Comma is not allowed</td>
</tr>
<tr>
<td>+5.0E3</td>
<td>Yes</td>
<td>(ANSI C supports unary plus)</td>
</tr>
<tr>
<td>3.5e-5</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>7.1e4</td>
<td>No</td>
<td>No white space is permitted</td>
</tr>
<tr>
<td>-4.5e-2</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>1.5E+2.5</td>
<td>No</td>
<td>Exponent must be an integer</td>
</tr>
<tr>
<td>$265</td>
<td>No</td>
<td>$ symbol is not permitted</td>
</tr>
<tr>
<td>0X7B</td>
<td>Yes</td>
<td>Hexadecimal integer</td>
</tr>
</tbody>
</table>

### Single Character Constants:

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character constants are:

\[
\text{‘5’, ‘X’, ‘&’}
\]

Note that the character constant ‘5’ is not the same as the number 5. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

```c
printf(“%d”, ‘a’);
```

would print the number 97, the ASCII values of the letter a. Similarly, the statement

```c
printf(“%c”, ‘97’);
```

would output the letter ‘a’.
String Constants:

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

“Hello”       “1987”       “WELL DONE”  “?!”       “5+3”        “X”

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example the symbol ‘\n’ stands for newline character. A list of such backslash character constants is given in table 2.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘\n’</td>
<td>audible alert (bell)</td>
</tr>
<tr>
<td>‘\t’</td>
<td>back space</td>
</tr>
<tr>
<td>‘\r’</td>
<td>form feed</td>
</tr>
<tr>
<td>‘\f’</td>
<td>new line</td>
</tr>
<tr>
<td>‘\v’</td>
<td>carriage return</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>vertical tab</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>single quote</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>double quote</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>question mark</td>
</tr>
<tr>
<td>‘\ ’</td>
<td>backslash mark</td>
</tr>
<tr>
<td>‘\0’</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 2. Backslash character constants

VARIABLES

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.

Examples: Averages, height, Total, Counter 1, etc.

As mentioned earlier, variable names may consist of letters, digits, and the underscore (_) character, subject to the following conditions:
1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first characters are treated as significant by many compilers.
3. Uppercase and lowercase are significant.
4. It should not be a keyword.
5. White space is not allowed.
DATA TYPES

C language is rich in its data types. ANSI C supports three classes of data types.

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

Primary Data Types

All C compilers support five fundamental data types, namely integer (int), character (char), floating point (float), double-precision floating point (double) and void. Many of them also offer extended data types such as long int and long double. Various data types and the terminology used to describe them are given in Fig 4.

![Diagram of primary data types in C](image)

Figure. 4 primary data types in C.

Table 3. Size and Range of Basics Data Types on 16–bit Machines

<table>
<thead>
<tr>
<th>Data type</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>−128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>−32,768 to 32,767</td>
</tr>
<tr>
<td>float</td>
<td>3.4e−38 to 3.4e+38</td>
</tr>
<tr>
<td>double</td>
<td>1.7e−308 to 1.7e+308</td>
</tr>
</tbody>
</table>

Integer Types
Integers are whole numbers with a range of value supported by a particular machine. The integers occupy one word of storage, and since the word sizes of machine vary (typically 16 to 32 bits) the size of the integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to range (-32768 to +32767) \(-2^{15}\) to \(+2^{15}-1\). The signed integer uses one bit for sign and 15 bits for the magnitude of the number.

![Figure 2.5 Integer types](image)

The C has three classes of integer storage, namely **short int**, **int** and **long int**, in both signed and unsigned forms. As in fig 5 **short int** represents fairly small integer values and require half the amount of storage as a regular **int** number uses. The unsigned integers use all the bits for the magnitude of the number and are always positive. The range of unsigned integer number will be from 0 to 65,535. Below Table 6 shows all the allowed combinations of basic types and qualifiers and their size range on a 16-bit machine.

**Table 6 Size and range of data types on 16 bit machine**

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

**Floating point Types**

Floating point (or real) numbers are stored in 32 bits (on 16 and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by keyword float. To get more accuracy the type double can be used to define the number. A double data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers.
FIGURE Floating point types

Void types

The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function.

Character Types

A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned char has values between 0 and 255, signed char has values from -128 to 127.

DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

- It tells the compiler what the variable name is.
- It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with it type. The syntax for declaring a variable is as follows:

data-type v1,v2,v3,.....vn;

v1,v2,v3,.....vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. Table 8 shows various data types and their equivalents.

Example:

int count;
int number, total;
double ratio;

Table 8 Data types and their Keywords
User-defined Type Declaration

Type Definition

C supports a feature known as “type definition” that allows users to define an identifier that would represent an existing data type. The user-defined type identifier can later be used to declare variables. It takes the general form:

```c
typedef type identifier;
```

where `type` refers to an existing data type and “identifier” refers to the “new” name given to the data type.

Example:

```c
typedef int units;
typedef float marks;
```

Here, `units` symbolizes `int` and `marks` symbolizes `float`. They can be later used to declare variables as follows:

```c
units E1, E2,
marks student1, student2;
```

`E1` and `E2` are declared as `int` variable and `student1` and `student2` are declared as floating point variables.
Enumerated Data Type

The enumerated data type is defined as follows:

```c
enum identifier {value1, value2, ... valuen};
```

The “identifier” is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces known as enumeration constants. Example:

```c
enum day {Monday, Tuesday, ... Sunday};
enum day week_start, week_end;

week_start = Monday;
```

the compiler automatically assigns integer digits beginning with 0 to all the enumeration constants.

**DECLARATION OF STORAGE CLASS**

Variables in C can have not only dat type but also storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```c
int m;
main()
{
    int i;
    float balance;
    function();
}
Function1()
{
    int i;
    float sum;
    ....
    ....
}
```

The variable `m` which has been declared before the main is called global variable. It can be used in all the functions in the program. A global variable is also known as an external variable.
The variables i, balance and sum are called local variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. The variable i has been declared in both the functions any change in the value of i in one function does not affect its value in other.

Storage class specifiers:

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concepts of scope and lifetime are important only in multiplication and multiple file programs. There are four storage class specifiers(auto, register, static, and extern) whose meanings are given in Table 10.

The storage class is another qualifier that can be added to a variable declaration as shown below:

```c
auto int count;
register char ch;
estatic int x,
extern long total.
```

Static and external (extern) variables are automatically initialized to zero. Automatic (auto) variables contain undefined values known as garbage unless they are initialized explicitly.

ASSIGNING VALUES TO VARIABLES

- Assignment statement
- Reading data from keyboard

Variables are created for use in program statements such as,

```c
Value=amount +inrate * amount;
while(year <= PERIOD)
{
    ....
    ....
    year= year + 1;
}
```
In the first statement, the numeric values stored in the variable \texttt{inrate} is multiplied by the value stored in \texttt{amount}. The result is stored in the variable \texttt{value}. This process is possible only if the variables \texttt{amount} and \texttt{inrate} have already been given values. The variable \texttt{value} is called target variable.

\textbf{Assignment Statement}

Values can be assigned using the assignment operator = as follows:

\texttt{Variable\_name = constant;}

Example:

\begin{verbatim}
    initial\_value = 0;
    final\_value = 100;
    balance = 50000;
    grade = 'A';
\end{verbatim}

\texttt{C} permits multiple assignments in one line:

Example:

\begin{verbatim}
    initial\_value = 0; final\_value = 100;
\end{verbatim}

During assignment operation, \texttt{C} converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

\textbf{INITIALIZATION:}

The process of giving initial values to variables is called initialization. Assigning the values at the time of variable is declared this is done with the following form:

\texttt{data\_type variable = constant;}

Examples:

\begin{verbatim}
    int final\_value = 100;
    char grade = 'A';
\end{verbatim}

\texttt{C} permits the initialization of more than one variable in one statement using multiple assignment operators.
Example: \( p = q = s = 0; \)

The following program shows declarations, assignments and values stored in various types of variables.

```c
main()
{
    /* ........ DECLARATIONS .................................... */
    float x, p;
    double y, q;
    unsigned k;

    /* ........ DECLARATIONS AND ASSIGNMENTS ....................... */
    int m = 54321;
    long int n = 1234567890;

    /* ........ ASSIGMENTS ...................................... */
    x = 1.2345678900000;
    y = 9.87654321;
    k = 54321;
    p = q = 1.0;

    /* ........ PRINTING ...................................... */
    printf("m = %d\n", m);
    printf("n = %ld\n", n);
    printf("x = %12lf\n", x);
    printf("x = %12f\n", x);
    printf("y = %12lf\n", y);
    printf("y = %12f\n", y);
    printf("k = %u p = %f q = %.12lf\n", k, p, q);
}
```

Reading Data from Keyboard
Another way of giving values to variables is input data through keyboard using the scanf function. It is a general input function available in C and is very similar in concept to the printf function. The general format of scanf is as follows:

    scanf("control string", &variable1, &variable2);

The control string contains the format of data being received. The ampersand symbol & before each variable name is an operator that specifies the variable name’s address.

Example:

    scanf("%d", &number),

when this statement is encountered by the computer, the execution stops and waits for the value of the variable number to be typed. The control string %d specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. The scanf provides an interactive feature and makes the program “user friendly”.

Program 10……

DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program.

Example: 3.142, representing the value of the mathematical constant “pi”.

Modifiability

we may like to change the value of “pi” from 3.142 to 3.14159 to improve accuracy of calculations. In this case we need to search throughout the program and explicitly change the value of the constant whenever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in the program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the “pass marks” at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

A constant is defined as follows:

    #define symbolic_name value_of_constant
Examples:

#define STRENGTH 100
#define PASS_MARK 50
#define PI 3.14159

Symbolic names are sometimes called constant identifiers. Since the symbolic names are constants not variables, they do not appear in declarations. `#define` statement is a preprocessor compiler directive. The following rules apply to a `#define` statement which define a symbolic constant:

- Symbolic names have the same form as variable names. Symbolic names are written in CAPITALS to distinguish them from normal variable names.
- No blank space between the pound sign `#` and the word `define` is permitted.
- `#` must be the first character in the line.
- A blank space is required between `#define` and symbolic name and between the symbolic name and the constant.
- `#define` statements must not end with a semicolon.
- After definition, the symbolic name should not be assigned any other value within the program by using an assignment statement. Example: `STRENGTH= 200;` is illegal.
- Symbolic names are NOT declared for data types.
- `#define` statements may appear anywhere in the program but before it is referenced in the program.

DECLARE VARIABLE AS CONSTANT

When we want the value of certain variables to remain constant during the execution of a program. We can do this by declaring the variables with the qualifier `const` at the time of initialization.

Example: `const int class_size=40;`

The `const` is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the int variable `class_size` must not be modified by the program.

DECLARE VARIABLE AS VOLATILE

The ANSI standard defines another qualifier `volatile` that could be used to tell explicitly the compiler that a variable’s value may be changed at any time by some external sources.

Example:
volatile int date;

the value of the date may be altered by some external factors even if it does not appear on the left hand side of an assignment statement. When we declare a variable as volatile, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

**OVERFLOW AND UNDERFLOW OF DATA**

The problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since the floating-point values are rounded off to a number of significant digits allowed, an overflow normally results in the largest possible real value, whereas an underflow results in zero. Integers are always exact within the limits of the range of the integral data types used. The overflow which is a serious problem may occur if the data type does no match the value of the constant.

*CHAPTER 3: Operators and Expressions*

1 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 3.1. The operators +,-,*, and / all work the same way as they do in the other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiples its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition or unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or unary minus</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo division</td>
</tr>
</tbody>
</table>

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

\[
\begin{align*}
a - b & \quad a + b \\
a * b & \quad a \div b \\
a \% b & \quad - a * b
\end{align*}
\]
Here \(a\) and \(b\) are variables and are known as *operands*. The modulo division operator \(\%\) cannot be used on floating point data. Note that C does not have an operator for exponentiation. Older versions of C does not support unary plus but ANSI C supports it.

### Integer Arithmetic

When both the operands in a single arithmetic expression such as \(a+b\) are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer values depends on the machine, as pointed out earlier. In the above examples, if \(a\) and \(b\) are integers, then for \(a=14\) and \(b=4\) we have the following results:

\[
\begin{align*}
a  \quad b &= 10 \\
a + b &= 18 \\
a \times b &= 56 \\
a \div b &= 3 \text{ (decimal part truncated)} \\
a \% b &= 2 \text{ (remainder of division)}
\end{align*}
\]

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is,

\[
\begin{align*}
-6 \div 7 &= 0 \quad \text{and} \quad -6 \div -7 = 0
\end{align*}
\]

But \(-6 \div -7\) may be zero or \(-1\). (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is,

\[
\begin{align*}
-14 \% 3 &= 2 \\
-14 \% -3 &= -2 \\
14 \% -3 &= -2
\end{align*}
\]

The program below shows the use of integer arithmetic to convert a given number of days into months and days.
```c
Program
main ()
{
    int months, days ;

    printf("Enter days\n") ;
    scanf("%d", &days) ;

    months = days / 30 ;
    days = days % 30 ;
    printf("Months = %d Days = %d", months, days) ;
}
```

**Output**

Enter days
265
Months = 8 Days = 25
Enter days
364
Months = 12 Days = 4
Enter days
45
Months = 1 Days = 15

The variables *month* and *days* are declared as integers. Therefore, the statement

```
months = days / 30;
```

truncates the decimal part and assigns the integer part to *months*. Similarly, the statement

```
days = days % 30;
```

assigns the remainder part of the division to *days*. Thus the given number of *days* is converted into an equivalent number of *months* and *days* and the result is printed as shown in the output.

**Real Arithmetic**

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the numbers of significant digits permissible, the final value is an approximation to the correct result.

If *x*, *y*, and *z* are *floats*, then we will have:
\[
x = 6.0 / 7.0 = 0.857143 \\
y = 1.0 / 3.0 = 0.333333 \\
z = -2.0 / 3.0 = -0.666667
\]

The operator \% cannot be used with real operands.

**Mixed-mode Arithmetic**

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

\[
15 / 10.0 = 1.5
\]

Whereas

\[
15 / 10 - 1
\]

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

**2 RELATIONAL OPERATORS**

We often compare two quantities depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol ‘<’, meaning ‘less than’. An expression such as

\[
a < b \text{ or } 1 < 20
\]

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false*.

For example

\[
10 < 20 \text{ is true}
\]

but

\[
20 < 10 \text{ is false}
\]

C supports six relational operators in all. These operators and their meanings are shown in Table 3.2.
Table 3.2 Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
</tr>
</tbody>
</table>

A simple relational expression contains only one relational operator and takes the following form:

```
<operator> <ae-1> <operator> <ae-2>
```

`ae-1` and `ae-2` are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

- `4.5 <= 10`  TRUE
- `4.5 <= 10`  FALSE
- `-35 >= 0`   FALSE
- `10 < 7 + 5` TRUE

where `a + b = c + d`  TRUE  only if the sum of values of `a` and `b` is equal to the sum of values of `c` and `d`.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

```
>    is complement of   <=
<    is complement of   >=
==   is complement of   !=
```

We can simplify an expression involving the `not` and the `less than` operators using the complements as shown below:

<table>
<thead>
<tr>
<th>Actual one</th>
<th>Simplified one</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
!(x < y)  x >= y
!(x > y)  x <= y
!(x != y) x == y
!(x <= y) x > y
!(x >= y) x < y
!(x == y) x != y

3 LOGICAL OPERATORS

In addition to the relational operators, C has the following three logical operators.

```
&&  meaning logical AND
||  meaning logical OR
!   meaning logical NOT
```

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

```
a > b && x == 10
```

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression. Like the simple relational expressions, a logical expression also yields a value of one or zero according to the truth table shown in the Table 3.3. The logical expression given above is true only if a > b is true and x == 10 is true, if either (or both) of them are false, the expression is false.

**Table 3.3 Truth Table**

<table>
<thead>
<tr>
<th>op-1</th>
<th>op-2</th>
<th>Value of the expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-zero</td>
<td>Non-zero</td>
<td>1</td>
</tr>
<tr>
<td>Non-zero</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Non-zero</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Some examples of the usage of logical expressions are:

1. if (age> 55 && salary <1000)
2. if (number < 0 || number > 100)
4 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, ‘=’. In addition, C has a set of ‘shorthand’ assignment operators of the form:

\[ v \ op= \ exp; \]

Where \( v \) is a variable, \( exp \) is an expression and \( op \) is a C binary arithmetic operator. The operator \( op= \) is known as the shorthand assignment operator.

The assignment statement

\[ v \ op= \ exp; \]

is equivalent to

\[ v = v \ op \ (exp); \]

with \( v \) evaluated only once. Consider an example

\[ x += y+1; \]

This is the same as the statement

\[ x = x + (y+1); \]

The shorthand operator \( += \) means ‘add \( y+1 \) to \( x \)’ or ‘increment \( x \) by \( y+1 \)’. For \( y = 2 \), the above statement becomes

\[ x += 3; \]

and when this statement is executed, 3 is added to \( x \). If the old value of \( x \) is, say 5, then the new value of \( x \) is 8. Shorthand assignment operators are illustrated in Table 3.4.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Example/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>sum = 10; 10 is assigned to variable sum</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>sum += 10; This is same as sum = sum + 10</td>
</tr>
</tbody>
</table>

Table 3.4 Shorthand Assignments Operators
The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

**PROGRAM 3.2:** Program of Fig. 3.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *=

The program attempts to print a sequence of squares of numbers starting from 2. The statement

\[ a \ast= a; \]

which is identical to

\[ a = a \ast a; \]

Replaces the current value of \( a \) by its square.

**Program**

```c
main()
```
```c
{
    while( a < N )
    {
        printf("%d\n", a);
        a *= a;
    }
}
```

Output
2
4
16

**Fig. 3.2** Use of shorthand operator `*=

## 5 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. The increment and decrement operators:

- `++` and `--`

The operator `++` adds 1 to the operand, while `--` subtracts 1. Both are unary operators and takes the following form:

- `++m`, or `m++`,
- `--m`, or `m--`,
- `++m` is equivalent to `m = m + 1`, (or `m += 1`),
- `--m` is equivalent to `m = m - 1`, (or `m -= 1`),

We use the increment and decrement statements in for and while loops extensively.

Consider the following:

```c
m = 5;
y = ++m;
```

In this case, the value of `y` and `m` would be 6. Suppose, if we rewrite the above statements as

```c
m = 5;
y = m++;
```

then, the value of `y` would be 5 and `m` would be 6.

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.
Rules for ++ and -- Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new values of the variable.
- The precedence and associativity of ++ and -- operators are the same as those of unary + and unary -. 

6 CONDITIONAL OPERATOR

A ternary operator pair "??" is available in C to construct conditional expressions of the form

\[ \text{exp1} \quad ? \quad \text{exp2} : \quad \text{exp3} \]

where \( \text{exp1} \), \( \text{exp2} \) and \( \text{exp3} \) are expressions.

The operator ? : works as follows: \( \text{exp1} \) is evaluated first, if it is nonzero (true), then the expression \( \text{exp2} \) is evaluated and becomes the value of the expression. If \( \text{exp1} \) is false, \( \text{exp3} \) is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either \( \text{exp2} \) or \( \text{exp3} \)) is evaluated. For example, consider the following statements

\[
\begin{align*}
\text{a} &= 10; \\
\text{b} &= 15; \\
x &= (a>b) \; ? \; a \; : \; b;
\end{align*}
\]

In this example, \( x \) will be assigned the value of \( b \). This can be achieved using the if..else statements as follows:

\[
\begin{align*}
\text{if} \; (a > b) \\
&\quad x = a; \\
\text{else} \\
&\quad x = b;
\end{align*}
\]

7 BITWISE OPERATORS

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting
them right or left. Bitwise operators may not be applied to float or double. Table 3.5 lists the bitwise operators and their meanings.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>bitwise exclusive OR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
</tbody>
</table>

### 8 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators (, and ->)

#### The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the value of the combined expression. For example, the statement:

```plaintext
value = ( x = 10, y=5, x+y );
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (i.e. 10 + 5) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

**In for loops:**

```plaintext
for ( n=1, m=10, n<=m, n++, m++ )
```

#### The sizeof Operator

The sizeof is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

```plaintext
m = sizeof(sum);

n = sizeof(long int);

k = sizeof(235L);
```
The `sizeof` operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

**PROGRAM 3.3:** In Fig. 3.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

```c
Program

main()
{
    int a, b, c, d;
    a = 15;
    b = 10;
    c = ++a - b;
    printf("a = %d b = %d c = %d\n", a, b, c);
    d = b++ +a;
    printf("a = %d b = %d d = %d\n", a, b, d);
    printf("a/b = %d\n", a/b);
    printf("a%b = %d\n", a%b);
    printf("a *= b = %d\n", a*=b);
    printf("%d\n", (c>d) ? 1 : 0);
    printf("%d\n", (c<d) ? 1 : 0);
}

Output

a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

9 **ARITHMETIC EXPRESSIONS**

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6.
Table 3.6  Expressions

<table>
<thead>
<tr>
<th>Algebraic expression</th>
<th>C expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax + bx - c</td>
<td>a * b - c</td>
</tr>
<tr>
<td>(m+n)(x+y)</td>
<td>(m+n) * (x+y)</td>
</tr>
<tr>
<td>ab</td>
<td>a * b/c</td>
</tr>
<tr>
<td>c</td>
<td>3 * x * x + 2 * x + 1</td>
</tr>
<tr>
<td>3x^2 + 2x + 1</td>
<td>x/y + c</td>
</tr>
<tr>
<td>(x/y) + c</td>
<td></td>
</tr>
</tbody>
</table>

10 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form:

```
Variable = expression;
```

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are:

```
x = a * b - c;
y = b / c * a;
z = a - b / v + d;
```

The blank space around an operator is optional and adds only to improve readability.

PROGRAM 3.4: The program in Fig. 3.4 illustrates the use of variables in expressions and their evaluation.

```
Program

main()
{
    float a, b, c, x, y, z;
}
```
11 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority: */%

Low priority: +

The basic evaluation procedure includes ‘two’ left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 3.4:

\[ x = a - b / 3 + c * 2 - 1 \]

When \( a = 9, b = 12, c = 3 \), the statement becomes

\[ x = 9 - 12 / 3 + 3 * 2 - 1 \]

and is evaluated as follows

**First pass**

Step 1: \( x = 9 - 4 + 3 * 2 - 1 \)
Step 2: \( x = 9 - 4 \div 6 - 1 \)

**Second pass**

Step 3: \( x = 5 + 6 - 1 \)

Step 4: \( x = 11 - 1 \)

Step 5: \( x = 10 \)

These steps are illustrated in Fig. 3.5. The numbers inside parenthesis refer to step numbers.

![Fig. 3.5 illustration of hierarchy of operations](image)

However, the order of evaluation can be changed by introducing parenthesis into an expression. Consider the same expression with parenthesis as shown below:

\[
9 - \frac{12}{(3 + 3)} \times (2 - 1)
\]

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

**First pass**

Step 1: \( 9 - \frac{12}{6} \times (2 - 1) \)

Step 2: \( 9 - \frac{12}{6} \times 1 \)

**Second pass**

Step 3: \( 9 - 2 \times 1 \)

Step 4: \( 9 - 2 \)

**Third pass**
Step 5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e., equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. For example:

\[ 9 \ (12 / (3 + 3) * 2) - 1 - 4 \]

Whereas

\[ 9 - ((12 / 3) + 3 * 2) - 1 = -2 \]

While parentheses allow us to change the order of priority, we may also use them to improve the understandability of the program.

**Rules for Evaluation of Expression**

- First, parenthesized sub expressions from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

Program for the following expression: \( a = 5 \ll 8 \&\& 6 \ll 5 \)

```c
void main()
{
    int a;
    a = 5<<8 \&\& 6<<5;
    printf("%d", a);
}
```

**OUTPUT: 1**

12 SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:
a = 1.0 / 3.0;

b = a * 3.0;

We know that (1.0 / 3.0) * 3.0 is equal to 1. But there is no guarantee that the value of b computed in a program will be equal to 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid a division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that the operands are of the correct type and range, and the result may not produce any overflow or underflow.

**PROGRAM 3.6:** Output of the program in Fig. 3.7 shows round-off errors that can occur in computation of floating point numbers.

```c
main()
{
    float sum, n, term;
    int count = 1;
    sum = 0;
    printf("Enter value of n\n");
    scanf("%f", &n);
    term = 1.0/n;
    while( count <= n )
    {
        sum = sum + term;
        count++;
    }
    printf("Sum = %f\n", sum);
}
```

**Output**

Enter value of n
99
Sum = 1.000001
Enter value of n
143
Sum = 0.999999

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation the result is not always 1.
13 TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as implicit type conversion.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the ‘lower’ type is automatically converted to the ‘higher’ type before the operation proceeds. The result is of the ‘higher’ type. A typical type conversion process is illustrated in Fig. 3.8.

Given below is the sequence of rules that are applied while evaluating expressions. All short and char are automatically converted to int, then

1. If one of the operands is long double, the other will be converted to long double and the result will be long double;
2. Else, if one of the operands is double, the other will be converted to double and the result will be double;
3. Else, if one of the operands is float, the other will be converted to float and the result will be float;
4. Else, if one of the operands is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int;
5. Else, if one of the operands is long int and the other is unsigned int, then
   (a) If unsigned int can be converted to long int, the unsigned int operand will be converted to long int;
(b) Else, both operands will be converted to \texttt{unsigned long int} and the result will be \texttt{unsigned long int};

6. Else, if one of the operands is \texttt{long int}, the other will be converted to \texttt{long int} and the result will be \texttt{long int};

7. Else, if one of the operands is \texttt{unsigned int}, the other will be converted to \texttt{unsigned int} and the result will be \texttt{unsigned int}.

\textbf{Conversion Hierarchy}

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below.

Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.

However, the following changes are introduced during the final assignment.
1. \text{float} \to \text{int} \text{causes truncation of the fractional part.}
2. \text{double} \to \text{float} \text{causes rounding of digits.}
3. \text{long} \to \text{int} \text{causes dropping of the excess higher order bits.}

**Explicit Conversion**

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

\[
\text{ratio} = \frac{\text{female\_number}}{\text{male\_number}}
\]

Since \text{female\_number} and \text{male\_number} are declared as integers in the program, the decimal part of the result of the division would be lost and \text{ratio} would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

\[
\text{ratio} = (\text{float}) \frac{\text{female\_number}}{\text{male\_number}}
\]

The operator \text{(float)} converts the \text{female\_number} to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of the result.

Note that in no way does the operator \text{(float)} affects the value of the variable \text{female\_number}. And also, the type of \text{female\_number} remains as \text{int} in the other parts of the program.

The process of such a local conversion is known as \text{explicit conversion} or \text{casting a value}. The general form of a cast is:

\[
\text{\textit{type-name}} \text{ expression}
\]

where \text{\textit{type-name}} is one of the standard C data types. The expression may be a constant, variable or an expression. Example

\[
x = (\text{int}) 7.5 \text{; here } 7.5 \text{ is converted to integer by truncation.}
\]

Casting can be used to round-off a given value. Consider the following statement:

\[
x = (\text{int}) (y + 0.5);
\]

If \(y\) is 27.6, \(y + 0.5\) is 28.1 and on casting, the result becomes 28, the value that is assigned to \(x\). Of course, the expression, being cast is not changed.

**PROGRAM 3.7:** Figure 3.9 shows a program using a cast to evaluate the equation:
main()
{
    float sum;
    int n;
    sum = 0;
    for(n = 1; n <= 10; ++n)
    {
        sum = sum + 1/(float)n;
        printf("%2d %6.4f\n", n, sum);
    }
}

Output
1 1.0000
2 1.5000
3 1.8333
4 2.0833
5 2.2833
6 2.4500
7 2.5929
8 2.7179
9 2.8290
10 2.9290

OPERATOR PRECEDENCE AND ASSOCIATIVITY

As mentioned earlier each operator, in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators at
the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from ‘left to right’ or from ‘right to left’, depending on the level. This is known as the associativity property of an operator. Table 3.8 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet been discussed.

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

```c
if (x == 10 + 15 && y < 10)
```

The precedence rules say that the **addition** operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

```c
if (x == 25 && y < 10)
```

The next step is to determine whether `x` is equal to 25 and `y` is less than 10. If we assume a value of 20 for `x` and 5 for `y`, then

- `x == 25` is FALSE (0)
- `y < 10` is TRUE (1)

The operator `<` enjoys a higher priority compared to `==`, `y < 10` is tested first and then `x == 25` is tested.

Finally we get: `if (false && true)`

Because one of the conditions is false, the complex condition is false.

**Rules of precedence and Associativity**

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied.

**Mathematical functions**

Mathematical functions such as cos, sqrt, log, etc. Are frequently used in analysis of real-life problems. Most of the C compiles support these basic math functions. To use these functions in a program, we should include the line:

```c
#include <math.h> in the beginning of the program.
```