Dear Friends,

Nepal Students Union, Pulchowk Campus Unit has been conducting different educational, sports, health oriented and other different activities. As a continuation of such programs we are sending you this material to assist in your studies.

If you have any suggestions and comment to our activities please feel free to write us at out email id: nsu.pulchowk@gmail.com

Also if you need any study materials please write us we will try our best to provide you what you need.

JAY NEPAL!!!!

Nepal Students Union
Pulchowk Campus Unit

Learning C by examples

(For all IT students)

Pramod Pageni

065/BEL/319

By:
Krishna Kandel,
(Lecturer, Kathmandu Engineering College,
Kalimati, Kathmandu, Nepal)

Baisakh, 2064
All rights reserved

Table of contents

Chapter No.	Title	Page No.
1.0	Introduction to computer	1
2.0	Problem solving using computers	12
3.0	Introductory features of C	23
4.0	Input and output	45
5.0	Structured programming fundamentals	53
6.0	Functions	77
7.1	Arrays	~~102~~ 91
7.2	Pointers	~~109~~ 102
7.3	Dynamic memory allocation	~~144~~ 109
7.4	Strings	118
8.1	Structures	133
8.2	Unions	141
9.0	File input/output	143
10.0	Features of C preprocessor	158

Chapter 1
Introduction to computer

1.1 Overview
Nothing epitomizes modern life better than the computer. For better or worse, computers have infiltrated every aspect of our society. Today computers do much more than simply compute: supermarket scanners calculate our grocery bill while keeping store inventory; computerized telephone switching centers play traffic cop to Millions of calls and keep lines of communication untangled; and automatic teller machines (ATM) let us conduct banking transactions from virtually anywhere in the world. But where did all this technology come from and where is it heading? To fully understand and appreciate the impact computers have on our lives and promises they hold for the future, it is important to understand their evolution.

1.2 Introduction to computer
A functional unit that can perform substantial computations, including numerous arithmetic and logic operations without human intervention during a run. In information processing, the term computer usually describes a digital computer. A computer may consist of a stand-alone unit or may consist of several interconnected units.

1.3 Historical development
A brief history of development of computers is summarized in the following points.

- History of computation was started when the people in the early Sumerian civilization started to keep records of transaction on clay table.(1200 BC)
- Actual computing using the machine started around 3000BC when Babylon invented the abacus.
- In 1642, Blaise Pascal created a gear driven adding machine named Pascalene which was the first mechanical adding machine.
- In 1673, Leibnitz developed a machine named "Stepped Reckoner" that could multiply.
- In 1801, Jacquard constructed a Loom, which was the first machine programmed with punched cards.
- In 1822, Charles Babbage designed and built his first difference Engine, which was credited the first mechanical computer. So, he is known as the " Father of Computer".
- In 1842, Ada Augusta Lovelace wrote the first program for the difference engine made by Babbage.
- In 1847,Babage designed his second version of the Difference Engine but he could not complete it.It was conceived in 1891 by the science museum in England.
- In 1854 George Bool developed Boolean Logic which is the system for symbolic and logical reasoning and basics for computer design.
- In 1890 Punched card machine was developed in USA.
- Before the development of the vacuum tube century, all the machines were mechanical.
- The first electronic computer, ENIAC (Electronic Numerical Integrators and Computer) was built in 1946.
- After this many machine like EDVAC (Electronic Discrete Variable Automatic Computer), IBM 701, 704 were built using transistors in stead of vacuum tube.
- IN 1961, Integrated Circuit(IC) were commercially available.
- Since then computers use ICs instead of individual transistors.
- IBM360 (International Business Machines) and DEC PDP 8 (Programmed Data Processor, Digital Equipment Corporation) were the most popular machine at that time.
- Altair 8800 was the first easily available PC(Personal Computer).
- In 1981, IBM lunched the PC.
- Since then, there has been significant development in the field of microelectronics and microprocessor.
- Altair 8800 was the first easily available PC.
- Today, we run a more than 2 GHz(GigaHertz.) processor on our desktop with more than 256 MB(MegaByte) of memory and more than 40 GB(GigaByte) of storage.

1.4 Generations of computers
A generation of computer refers to the state of improvement in the development of computer system. In terms of technological developments over time, computers have been broadly classified into five generations. The lines of distinction between each generation are not exact, and some overlap in technologies already being existed. A major technological development that fundamentally has changed the way computers operate, resulting in increasingly smaller, cheaper, more powerful and more efficient and reliable devices characterize each generation of computer.

1.4.1 First Generation Computers (1945-1956): vacuum tubes
The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions. First generation computers relied on machine language to perform operations, and they could only solve one problem at a time. Input was based on punched cards and paper tape, and output was displayed on printouts.
Examples: ENIAC, UNIVAC-I, UNIVAC-II, EDSAC, Harvard Mark I (electromechanical), UNIVAC Honeywell Datamatic 1000.
Where,

UNIVAC - UNIVersal Automatic Computer,
EDSAC - Electronic Delay Storage Automatic Calculator

1.4.2 Second Generation Computers (1956-1963): Transistors

Transistors replaced vacuum tubes in the second generation of computers. The transistor was invented in 1947 but did not see widespread use in computers until the late 50s. The transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors. Though the transistor still generated a great deal of heat that subjected the computer to damage, it was a vast improvement over the vacuum tube. They associated all the components like as the modern days computers e.g. Printers, tape storage, disk storage, memory, operating system and stored programs. Second generation computers moved from cryptic binary machine language to symbolic or assembly languages, which allowed programmers to specify instructions in words. High-level programming languages were also being developed at this time, such as early versions of COBOL and FORTRAN. These were also the first computers that stored their instructions in their memory, which moved from a magnetic drum to magnetic core technology.
Examples : UNIVAC 1107, UNIVAC III, IBM 7030 Stretch, IBM 7070, 7080, series, Honeywell 800, Honeywell 400 series.

1.4.3 Third Generation Computers (1964 - 1971) : Integrated Circuits (IC)

The development of the integrated circuit was the characteristic of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers. Instead of punched cards and printouts, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time with a central program that monitored the memory. Computers for the first time became accessible to a mass of people because they were smaller and cheaper than their predecessors. The operating system used in those computers allowed to run many different programs at once with a central program that monitored and coordinated the computers memory and other resources.
Examples: Burroughs 6700, Honeywell 200, IBM System/360, System 3, System 7, UNIVAC 9000 series, GE 600 series, GE 235
Where, GE - General Electric

1.4.4 Fourth Generation Computers (1971-Present) : Microprocessors

Computers in this generation used LSI, VLSI and ULSI chips. The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What in the first generation filled an entire room could now fit in the palm of the hand. The Intel 4004 chip, developed in 1971, located all the components of the computer - from the central processing unit and memory to input/output controls - on a single chip. In 1981 IBM introduced its first computer for the home user and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors. As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUIs, mouse and handheld devices.
Examples: Computers what we have been using today, IBM System 3090, IBM RISC 6000, Cray 2 etc.

1.4.5 Fifth Generation Computers (Present and Beyond): Artificial Intelligence

Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. High speed logic and memory chips, virtual reality generation, satellite links, high performance, micro miniaturization, knowledge-based platforms etc will be the other features of fifth generations of computers. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization. Some computers in the present generations have the characteristics of future generations such as expert system, voice recognitions, parallel processing etc.

1.5 Classification of computers

1.5.1 On the basis of working mode

- Analog computer
- Digital computer
- Hybrid computer

1.5.1.1 Analog computer

An analog computer is one that measures physical values such as temperature, pressure, humidity etc

1.5.1.2 Digital computer

A computer is digital which directly counts numbers (digits), letters or other special symbols. Digital computers are mostly used in statistics, mathematics, scientific calculation and mostly in engineering.

1.5.1.3 Hybrid computer

Hybrid computers can do the task of digital as well as analog.

1.5.2 On the basis of size

Micro computer
Minicomputer
Mainframe computer
Super computer

1.5.2.1 Micro computer

Generally a computer that resides in user's desktop is called a microcomputer. Microcomputers include PC's, laptops, notebooks and handheld computers.

1.5.2.2 Mini computers

- These computers are larger than the mini computers.
- 3-25 times faster then the microcomputers.

1.5.2.3 Mainframe computers

- These are larger then the mini computers.
- These are 10 to 100 times faster than the microcomputers.

1.5.2.4 Super computers

- These are the largest in size
- Most powerful computers
- Can process at speeds of several hundreds or even thousands of millions of instructions per seconds.

1.6 Introduction to number system

A number system is a set of numbers or number-like objects together with some arithmetic operations that can be performed on them. Types of number system are:

- Decimal number system
- Octal number system
- Hexadecimal number system
- Binary number system

1.6.1 Decimal number system

The decimal number system is the oldest positional number system. In position number system , a number is represented by a set of symbols. Each symbol represents a particular value depending on its position. The actual number of symbols used in a positional system depends on its base. The decimal system uses a base of 10. It uses 10 symbols: 0 to 9. Any number can be represented by arranging symbols in various positions. In this system, each position represents a specific power of 10. The position and place values of decimal system are as:

Position →	4	3	2	1	0
Place value →	10^4	10^3	10^2	10^1	10^0

For example decimal number 123 represents:
$1 * 10^2 + 2 * 10^1 + 3 * 10^0$
$=1 * 100 + 2 * 10 + 3 * 1$
$=100 + 20 + 3$
$=123$

1.6.2 Octal number system

The octal number system uses a base of 8. It uses 10 symbols: 0 to 7. Any number can be represented by arranging symbols in various positions. In this system, each position represents a specific power of 8. The position and place values of octal system are as:

Position →	4	3	2	1	0
Place value →	8^4	8^3	8^2	8^1	8^0

For example, the decimal equivalent of octal number 234 can be found as:
$2 * 8^2 + 3 * 8^1 + 4 * 8^0 =$
$2 * 64 + 3 * 8 + 4 * 1 =$
$128 + 24 + 4 =$
156

1.6.3 Hexadecimal number system

The octal number system uses a base of 16. It uses 16 symbols: 0 to 9 and A to F. A denotes 10, B denotes 11 and so on. Any number can be represented by arranging symbols in various positions. In this system, each position represents a specific power of 16. The position and place values of hexadecimal system are as:

Position →	4	3	2	1	0
Place value →	16^4	16^3	16^2	16^1	16^0

For example, the decimal equivalent of hexadecimal number B25 can be found as:
$B * 16^2 + 2 * 16^1 + 5 * 16^0$

=B * 256 + 2 * 16 + 5 * 1
=2816 + 32 + 5
=2853

1.6.4 Binary number system

The octal number system uses a base of 2. It uses 2 symbols: 0 and 1. Any number can be represented by arranging symbols in various positions. In this system, each position represents a specific power of 2. The position and place values of binary system are as:

Position →	4	3	2	1	0
Place value →	2^4	2^3	2^2	2^1	2^0

For example, the decimal equivalent of binary number 1101 can be found as:
$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
=1 * 8 + 1 * 4 + 0 * 2 + 1 * 1
=8+4+0+1
=13

Computers store and process numbers, letters and words that are often referred to as data. Since computers can not understand the Arabic numeral or English alphabet, we should use some codes that can easily to be understood by them. In all modern computers, storage and processing units are made of a set of silicon chips, each containing a large numbers of transistors. A transistor is a two state device that can be put "off" and "on" states by passing electric current through it. Since transistors are sensitive to currents and act like switches, we can communicate with the computers using electric signals, which are represented as series of **pulse** and **no pulse** conditions. For easy of use, a pulse is represented by code **1** and no pulse by code **0**. These are called **binary digits (bits)**. A series of 1's and 0's are used to represent number and characters. Which provide a way for human and computers to communicate with one another. The numbers represented by binary digits are called binary numbers. Computers not only store data in binary form but also process data in binary form. Although computers store information in binary, it becomes cumbersome to display large numbers. For this reason, internal contents of computers are displayed in hexadecimal, decimal and decimal. There are different types of binary number formats. These are discussed in subsequent topics.

1.6.4.1 The bit

The smallest "unit" of data on a binary computer is a single bit. A single bit is capable of representing only two different values such as zero or one, true or false, on or off or right or wrong etc.

1.6.4.2 The nibble

A nibble is a collection of bits on a 4-bit boundary. With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits.

b3	b2	b1	b0

Where b0 represents the bit at bit position 0. Similarly, b1, b2 and b3 represents the bit at their corresponding bit positions. Bit at position 0 (right most) is known as Least Significant Bit (LSB). Similarly, bit at the leftmost position is called Most Significant Bit (MSB). All the other bits are referred by their respective bit numbers.

1.6.4.3 The Byte

A byte consists of eight bits. The bits in a byte are numbered from bit zero (b0) through seven (b7) as follows:

b7	b6	b5	b4	b3	b2	b1	b0

Bit 0 is the low order bit or least significant bit, bit 7 is the high order bit or most significant bit of the byte. We'll refer to all other bits by their number.
A byte also contains exactly two nibbles. Bits b0 through b3 comprise the low order nibble, and bits b4 through b7 form the high order nibble. Since a byte contains eight bits, it can represent 2^8 or 256, different values. Generally, we will use a byte to represent:

- Unsigned numeric values in the range 0 to 255.
- Signed numbers in the range -128 to +127.
- ASCII (American Standard Code for Information Interchange) character codes. (see 3.4.3.3)
- Other special data types requiring no more than 256 different values.

1.6.4.4 The word

The boundary for a Word is defined as either 16-bits or the size of the data bus for the processor, and a Double Word is Two Words. Therefore, a Word and a Double Word is not a fixed size but varies from system to system depending on the processor. We will number the bits in a word starting from bit zero (b0) through fifteen (b15) as follows:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Like the byte, bit 0 is the LSB and bit 15 is the MSB. When referencing the other bits in a word use their bit position

form the high order byte. With 16 bits, we can represent 2^16 (65,536) different values. These could be the unsigned numeric values in the range of 0 to 65,535, signed numeric values in the range of -32,768 to +32,767 or any other data type with no more than 65,536 values. The three major uses for words are

- 16-bit integer data values
- 16-bit memory addresses
- Any number system requiring 16 bits or less

1.6.4.5 The double word

A double word is exactly what its name implies, two words. Therefore, a double word quantity is 32 bits. Naturally, this double word can be divided into a high order word and a low order word, four bytes or eight nibbles.
Double words can represent all kinds of different data. It may be

- An unsigned double word in the range of 0 to 4,294,967,295,
- A signed double word in the range -2,147,483,648 to 2,147,483,647,
- A 32-bit floating point value
- Any data that requires 32 bits or less.

1.6.4.6 Summary of binary number formats.

Table 1.1 illustrates the summary of binary number formats.

Table 1.1 Summary of binary number formats

Name	Size (bits)	Example
Bit	1	1
Nibble	4	0101
Byte	8	0000 0101
Word	16	0000 0000 0000 0101
Double Word	32	0000 0000 0000 0000 0000 0000 0000 0101

In any number base, we may add as many leading zeroes as we wish without changing its value. However, we normally add leading zeroes to adjust the binary number to a desired size boundary. In above example, binary number 101 is represented in different formats by adding zeroes. Summary of number systems is illustrated in table 1.2. Table 1.3 illustrates the decimal numbers 1 to 15 in their equivalent binary, decimal and hexadecimal form.

Table 1.2 Summary of number system

Name	Base	Symbol	Example
Binary	Base 2	B	1111B
Octal	Base 8	Q or O	17O
Decimal	Base 10	none or D	15
Hexadecimal	Base 16	H	0FH

Table 1.3 Illustration of numbers in different systems

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
00	0000B	00Q	00H	08	1000B	10Q	08H
01	0001B	01Q	01H	09	1001B	11Q	09H
02	0010B	02Q	02H	10	1010B	12Q	0AH
03	0011B	03Q	03H	11	1011B	13Q	0BH
04	0100B	04Q	04H	12	1100B	14Q	0CH
05	0101B	05Q	05H	13	1101B	15Q	0DH
06	0110B	06Q	06H	14	1110B	16Q	0EH
07	0111B	07Q	07H	15	1111B	17Q	0FH

1.7 Computer systems and organization

Most of the today's computers are digitatl and operate on binary system. Any digital computer stores data and instructuions in combinations of binary digits. The combinations of eight bits is a byte. A byte is the basic unit of data representation and processing in computer system. Modern computers are capale of processing more than one byte at a time e.g. 2,4,8 bytes.

There are two main components of a computer system:

- Hardware
- Software

1.7.1 Hardware

Hardware is called "hard" because, once it is built, it is relatively difficult to change . It is the physical part of a computer system which includes the parts which we can see and touch. Which are mechanical, magnetic, electronic and electrical components. These devices are capable of accepting and storing computer data, executing a systematic sequence of operations on computer data or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control or other logical functions. For example monitor, keyboards, mouse, different types of chips, processor, connecting wires, storage devices, nuts, bolts etc are the hardware components.

Computer hardware system organization

All the hardware devices in a computer system are organized as in the following figure.

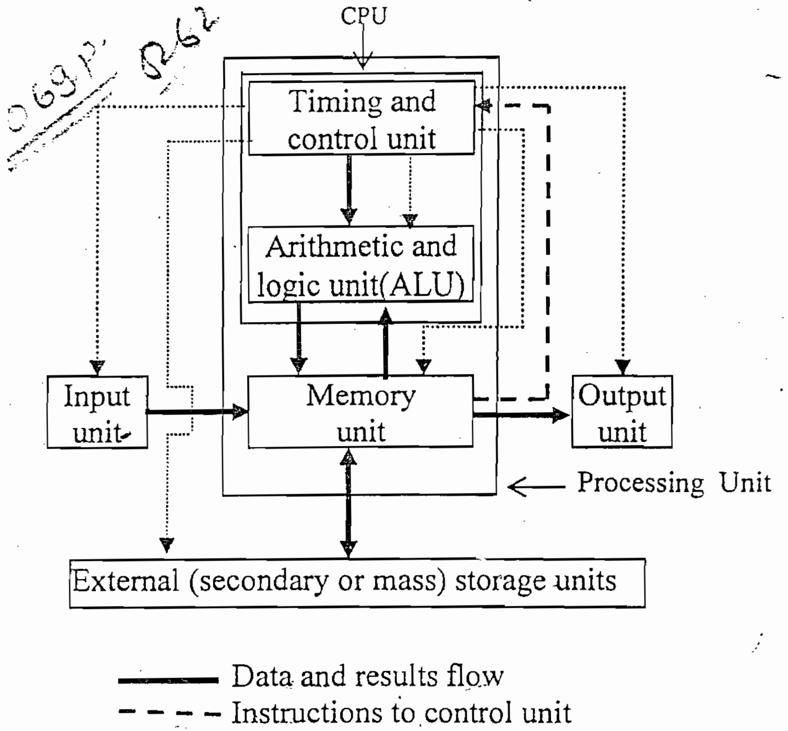


Fig. 1.2 Block diagram of a computer system

1.7.1.1 Input unit

This unit consists of electronic or electromechanical or both devices used to take information and data that are external to the computer and put them into memory unit or arithmetic and logic unit (ALU). The input unit is used to enter data and program into the memory unit and ALU prior to the starting of the computer and also during the execution of the program. The processing unit requires the data in the appropriately formatted electronic signals. Some communly used input devces are keyboard, mouse, joystick, digitizer, scanner, touchpanel, optical character recognization (OCR), magnetic ink character recognization(MICR) etc. Similarly, magnetic disks, optical disks, floppy disks and magnetic tapes can be used as input devices as well as storage deveces.

1.7.1.2 Processing unit (PU)

The processing module consists of two main parts:

- Central Processing Unit (CPU) or MPU (micro processing unit)
- Primary storage or main memory

a. Central Processing Unit (CPU) or MPU (micro processing unit)

The CPU is the major component of a computer; the "electronic brain" of the machine. It consists of the electronic circuits needed to perform operations on the data. The CPU is where most calculations take place. The two main parts of CPU are:

Arithmetic Logic Unit (ALU)

This unit is used to perform all arithmetic and logical operations such as addition, multiplication, comparison (decision) etc. All the logical decisions are done on the basis of "on" or "off" state representing true and false conditions. The comparison is related to logical operations such as AND, OR and NOT.

Control Unit (CU)

This unit co-ordinates the activities of all other units in the system. It implements the microprocessor instruction set. It extracts instructions from memory, decodes and executes them and sends the necessary signals to the ALU to perform the needed operations. It controls the transfer of data and in formations between various units. It is also used to initiate

appropriate actions by the arithmetic unit. Although the control section doesn't processes data, it acts as a central nervous system for the other data-manipulating components of the computer system.

The CPU lies inside a box called main machine or chassis on the motherboard. It is connected with ROM BIOS (Read Only Memory Basic Input Output System), hard disk, floppy disk, timing chips, control cards, expansion slots etc.

b. Primary storage or main memory

Main Memory is where programs that are currently being executed as well as their data are stored. The CPU fetches program instructions in sequence, together with the required data, from Main Memory and then performs the operation specified by the instruction. Once data are fed into primary storage from input devices, they are held and transferred. when needed, to the arithmetic – logic section, where the processing takes place. No processing occurs in primary storage. Intermediate results generated in the arithmetic –logic unit are temporarily placed in a designated working storage area until needed at a later time. Data may thus move many times before the processing is finished. Once completed, the final results are released to an output device. Information may be both read from and written to any location in Main Memory so the devices used to implement this block are called **random access memory** chips (RAM). The contents of Main Memory (often simply called **memory**) are both temporary (the programs and data reside there only when they are needed) and volatile (the contents are lost when power to the machine is turned off).

The purpose of using primary memory can be summarized in the following points.

- Data are fed into an input storage area where they are held until ready to be processed.
- A working storage space that is like a sheet of scratch paper is used to hold the data being processed and the intermediate results of such processing.
- An output storage area holds the finished results of the processing operations until they can be released.
- In addition to these data-related purposes, the primary storage section also contains a program storage area that holds the processing instructions.

The main memory is faster than secondary memory but expensive.

1.7.1.3 Secondary storage unit (mass storage, auxiliary storage)

The secondary storage unit provides more long term and stable storage for both programs and data. In modern computing systems, this secondary storage is most often implemented using rotating magnetic storage devices, more commonly called disks (though magnetic tape may also be used); therefore, Secondary Memory is often referred to as **the disk**. The physical devices making up secondary storage are also known as **mass storage devices** because relatively large amounts of data and many programs may be stored on them. Secondary storage also stores operating system, data files, compilers, assemblers, application programs etc. The CPU does not read information (residing in the secondary memory) directly from the secondary storage. The programs and data residing in the secondary storage, if needed by CPU are first transferred from the secondary storage to the primary memory. Then the CPU reads from the primary memory. The results are also stored in the secondary storage. The secondary storage is the mass storage. It is slower but cheaper then primary memory. Some examples of the secondary storage are hard disk, floppy disk, CD–ROM, DVDs, pen drives, magnetic tapes etc.

1.7.1.4 Output unit

In general, these units include devices provide the physical interface between the computer and its environment by allowing humans or even other machines to communicate with the computer. The responsibility of this unit is to transfer data and information from the computer to the "outside world". The output devices are directed by the control unit and can receive the data from memory or the ALU, which is then put into appropriate form for external use. The main softcopy output unit is Video Display Unit(VDU). Similarly, LCDs (Liquid Crystal Display), indicator lights, digital to analog converters (DAC) ete are the other softcopy output devices. Printers, plotters are the hard copy output devices. Similarly, magnetic disks, optical disks, floppy disks and magnetic tapes can be used as output devices as well as storage deveces.

1.7.2 Computer software

Software is defined as a set of coded commands (programs) that tell a computer what tasks to perform It is another part of a computer system. Without software a computer can not do anything. These programs are called software because these are relatively easy to change both the instructions in a particular program as well as which program is being executed by the hardware at any given time.

1.7.2.1 System software

It is a term referring to any computer software whose purpose is to help run the computer system. Most of it is responsible directly for controlling, integrating and managing the individual hardware components of a computer system. Specific kinds of system software include operating systems (OS), device drivers, programming tools etc. An operating system (OS) is a main system software It controls all parts of the computer system. The major functions of the OS are to handle all input devices, handle all output devices, coordinate and manage use of other resources like memory, disk, CPU etc., accept commands from users, provide an environment over which other programs (software) can run, transferring data from memory to disk or rendering text onto a display etc. DOS (disk operating system), Windows, Linux are some popular examples of operating system.

1.7.2.2 Application software

Application software is a subclass of computer software that employs the capabilities of a computer directly to a task that the user wishes to perform. Application softwares are designed to process data and support users in an organization such as solving equations or producing bills, result processing of campuses , data processing of accounts

very large number of business customers. These are specially designed and developed for end users. These run on top of the operating system. Examples include word processing software like Word/Word Perfect, spreadsheets like Excel or Lotus 123 etc. Application softwares can be classified into following categories.

- Tailored software.
- Packaged software.
- Utility software

Tailored software: These kinds of software are developed for solving a particular problem. For example, a software for processing of Institute Of Engineering only. The nature of processing of data seems similar but actually they are different in many ways. For different purposes different programs are to be written.

Packaged Software: Different kinds of programs related to an application are combined together to form a package. Those packages are common to many users. For examples following are some popular packages.
Microsoft Word, Word Perfect: There are word processing software for creating text-based documents such as news letter, browser, books etc.
Microsoft Excel, Quattro pro: These are spreadsheet application for creating numeric-based documents such as budgets and balance sheets.
Microsoft Access, Paradox: These are database management software for building and manipulating large sets of data.
Similarly Microsoft PowerPoint for electronic slide show, Photoshop for manipulation photograph, movies or animation, different types of games for entertainments etc.

Utility Software: These are special types of application software which helps us to fine tune the performance of a computer, prevent unwanted actions or perform system related tasks such as checking for virus and removing virus, sending fax, file copying, editors, system utilities which provide informations about the current state of the use of files, memory, users and peripherals, e.g., disk info, check disk, debuggers for removing "bugs" from programs.

1.8 Program

Basic commands that instruct the computer system to do something are called instructions. An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner is a program. A program is like a recipe. It contains a list of ingredients (called variables) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text or graphical images. Without programs, computers are useless.

1.9 Programming language

A programming language is a standardized communication technique for describing instructions for a computer. Each programming language has a set of syntactic and semantic rules used to define computer programs. A language enables a programmer to precisely specify what data a computer is to act upon, how these data are to be stored/transmitted and what actions are to be taken under various circumstances. Programming languages are another kinds of software which enables us to develop differet kinds of softwares. Programming languages are classified mainly in two categories on the basis of creating instrctions.

- Low level languages
- High level languages

1.9.1 Low level languages

These are much closer to hardware. Before cerating a program for a hardware, it is required to have through knowledge of that hardware. A program can not be run on different hardware. Low level languages are specific to hardware. Low level language is also divided in two types.

- Machine language
- Assembly language

1.9.1.1 Machine Language

Machine languages are the lowest-level programming languages. A computer understands programs written only in the machine language. While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers. Ultimately, machine code consists entirely of the 0's and 1's of the binary number system. Which are also called bits. Early computers were programmed using machine languages. Programs written in machine language are faster and efficient. Writing program in machine language is very tedious, time consuming, difficult to find bugs in longer programs.

1.9.1.2 Assembly Language

To overcome the difficulties of programming in machine language assembly languages were developed. An assembly language contains the same instructions as a machine language, but each instruction and variable have a symbol instead of being just numbers. These symbols are called mnemonic. For example if 78 is the number to add two numbers, ADD could be used to replace it. After developing assembly languages, it was easier to program using symbols instead of numbers. But the program written in assembly language must be converted to machine language which could be done by assembler. Each type of CPU(Central Processing Unit) has its own machine language and assembly language. So an assembly language program written for one type of CPU won't run on another. This shows that assembly language is also machine dependent and time consuming. Programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a high-level language. Programs written in machine

Assembler
An assembler is a program (software) which translates the program written in assembly language to machine language. Assembler produces one machine instruction for each source instruction and symbolic address to machine address. Assembler also includes the necessary linkage for closed subroutine, allocates areas of storage, detects and indicates invalid source language instructions. The object program is stored on secondary storage media. The resulting program can only be executed when the assembly process is completed.

1.9.2 High level language
The languages are called high level language if their syntax is closer to human language. High-level languages were developed to make programming easier. Most of the high level languages are English like languages. They use familiar English words, special symbols (!, & etc), and mathematical symbols (+,-,*,/ etc) in their syntax. Therefore, high-level languages are easier to read, write, understand and programming. Which is the main advantage of high-level language over low-level language. Each high level language has their own set of grammar and rules to represent set of instructions. Programming languages such as C, C++, Java, FORTRAN (acronym derived from The IBM Mathematical FORmula TRANslating System) or Pascal, Lisp, BASIC (Beginner's All-purpose Symbolic Instruction Code), Lisp, Prolog are some examples of high-level languages. These enable a programmer to write programs that are more or less independent of a particular type of computer. Like assembly language programs, programs written in a high-level language also need to be translated into machine language. This can be done either by a compiler or an **interpreter**. Before discussing about them let us know some terms.

1.10 Some terms
Source code
Initially, a programmer writes a program in a particular programming language like C, C++ and FORTAN etc. This form of the program is called the source program, or more generically, source code. Source code is the only format that is readable by humans. The source code consists of instructions in a particular language. When we purchase programs, we usually receive them in their machine-language format. This means that we can execute them directly, but we cannot read or modify them.

Object Code
The code produced by a compiler is called object code. It is an intermediary form. Object code is often the same as or similar to a computer's machine language. Object code need to be converted in to executable code using linkers.

Executable code
Code that is ready to run is called executable code or machine code.

Compile
To transform a program written in a high-level programming language (source code) into object code

Linker
Many programming languages allow us to write different pieces of code, called modules, separately. This simplifies the programming task because we can break a large program into small, more manageable pieces. Eventually, we need to put all the modules together. This is the job of the linker. Therefore, a linker is a program that combines object modules to form an executable program. In addition to combining modules, a linker also replaces symbolic addresses with real addresses. We need to link a program even if it contains only one module. A linker is also called a binder.

Link
In programming, the term link refers to execution of a linker.

Run
To execute a program.

1.11 Compiler
A program that translates *source code* into *object code*. The compiler looks at the entire portion of source code and reorganizes the instructions. Every high-level programming language (except strictly interpretive languages) comes with a compiler. A compiler defines the acceptable instructions. Because compilers translate source code into object code which is unique for each type of computer. Many compilers are available for the same language. For example, we have been running Turbo C compiler in windows environment and gcc compiler in Linux system. More than a dozen companies develop and sell C compilers for the PC.

1.11.1 Compilation process
The process of translation from high level language(source code) to low level languare(object code) is called **compilation**. In this process, Source code must go through several steps before it becomes an executable program. Which is illustrated in figure 1.2.

- The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code.
- The final step in producing an executable program is to pass the object code through a linker. The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.
- Compilation process ends by resulting an executable program.

- The compiler stores the object and executable files in secondary storage.
- If there is any illegal instruction in the source code, compiler lists all the errors during compilation.

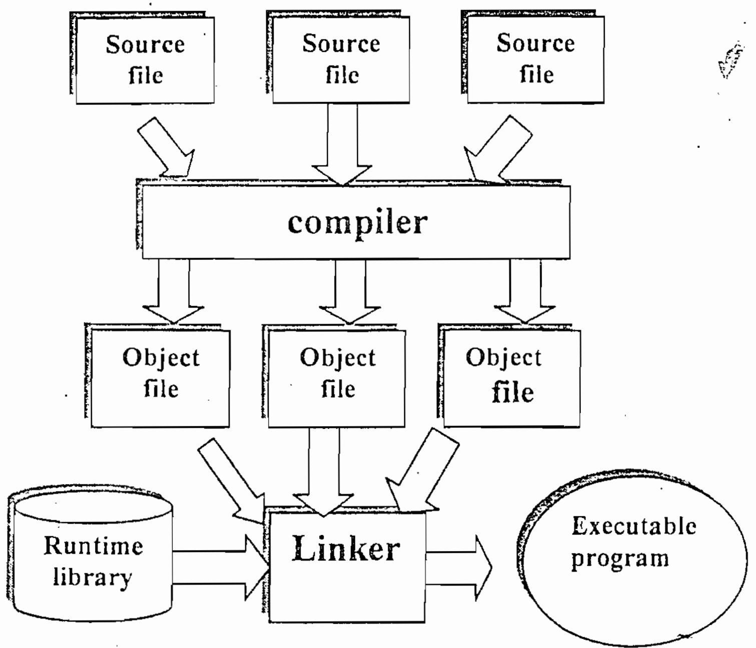


Figure 1.2: Illustration of Compilation process

1.12 Interpreter

An interpreter is a program which converts each high-level program statement in to machine code just before the program statement is to be executed. Translation and execution occur immediately one statement at a time. Interpreter directs the CPU to obey each program statement at a time.

Comparison between compiler and interpreter

S.No.	Compiler	Interpreter
1.	Compiler converts all the statements in source to object code and finally in executable code resulting an exe file.	Where as interpreter converts each statement before executing it. It does not produce an exe file.
2.	Compilers require some time before producing an executable program.	Interpreters can execute a program immediately.
3.	Once compiled program does not need to recompile for next run. Therefore, executable programs produced by compilers run much faster and efficient.	For running next time, it is required to repeat the process from the beginning. So it is slower process.
4.	Immediate editing and executing a program is costly process because it takes long time for compilation process if the program is long.	It is possible to execute the edited program immediately. Therefore, it is can be used in software developing phase and in learning phase for students.
5.	C,C++, FORTAN are examples of compiled language.	BASIC, LISP are interpreted languages.

Language translators compiler, interpreter and assemblers are also called language processors.

1.13 Firmware

Firmware is software that is embedded in a hardware device e.g. in a read-only memory (ROM) chip, erasable programmable read-only memory (EPROM) chip or as a binary image file that can be uploaded onto existing hardware by a user by using special external hardware. The BIOS in IBM-compatible personal computers is an example of firmware.

Exercise 1

1. What do you understand by Computer Generation? List salient features of each generation. [TU 2058/p/1-a, 059/c/1-a, 2062/b/1-a, similar to 2063/b-1]
2. Draw the block diagram of a computer and explain the function of each block.[060/p/1-a, 062/p/1]

3. Define programming language? Explain its types. [059/p/1-a]
4. What do you mean by computer software? Describe briefly about system software. [057/c/1-a]
5. Write down the difference between compiler and interpreter. Explain Compilation process in detail. [2061/p/1-b, 2062/p/3]
6. How many ways can we convert high-level languages to the machine-level language? Explain all in detail with block diagram. [2063/p/1]
7. What is the difference between source code and object code?

Note:
2058/p/1-a → 2058/Poush, question number 1, part a, in TU, IOE exam
059/c/1-a → 2058/Chaitra, question number 1, part a, in TU, IOE exam
2062/b/1-b → 2062/Baishakh, question number 1, part b, in TU, IOE exam
2063/b/1 → 2063/ Baishakh, question number 1, in TU, IOE exam

Chapter 2
Problem solving using computers

Different kinds of general purpose software packages are available in the market. But, sometimes these packages cannot fulfill all clients'(users') requirements. So to fulfill the client's requirements new software must be developed. Generally, the clients come to the developer's site with their problems to be solved and finding a computer based solution. A computer-based solution will be software to fulfill the clients' requirements. To develop good quality software, it is required to go through the following phases.

2.1 Problem analysis

It is important to give a clear, concise problem statement. It is also called problem definition. The problem definition should clearly specify the following tasks.

a. **Objectives:** The problem should be stated clearly so that there will not be the chance of having right solution to the wrong problem. Simple program can be stated easily and early but for complex problem may need a complex analysis with careful coordination of people, procedures and programs.

b. **Output requirements:** Before we know what should go into the system, we must know what should come out from the system being developed. Although the system analyst or programmer, the best person to design output is the end user. So, it is the better way of designing output sitting with the end uses.

c. **Input requirements:** To get the above designed output, it is required to define the input data and source of input data. For example, in a student information system, input data may be students' records and source may be college administration.

d. **Processing requirements:** It is required to clearly defined processing requirements to convert the given input data to the required output. In processing requirements, there may be hardware platform, software platform, manpower etc.

e. **Evaluating Feasibility:** It is one of the important phases where we manly decide whether the proposed software development task is technically and economically feasible.

Finally there should be a proper documentation of the different tasks of problem analysis done in the above stages.

2. 2 Algorithm development and flowcharting

2.2.1 Algorithm

An algorithm is defined as a set of ordered steps or procedures necessary to solve a problem. To be an algorithm, the set of steps must be unambiguous and have a clear stopping point. Each step tells what task is to be performed. Algorithms can be expressed in any language e.g. English. It is a verbal form of a program. Let us take an example of writing a letter using Microsoft word.

a. Start
b. Place the switch of all attached devices in ON state.
c. To start computer, place the switch of system unit in ON state.
d. Wait until the window's desktop appears on the screen.
e. Go to start button.
f. Click on the programs on the start menu
g. Click on the Microsoft word icon on start menu.
h. Wait until the window of Microsoft word appears.
i. Type a letter to your friend.
j. Save it giving a file name.
k. Close the window of Microsoft word.
l. Go to start button.
m. Click on the shutdown button.
n. Wait until the message " It is now safe to turn off your computer" appears.
o. Place the switch of system unit in OFF state.
p. Stop.

While writing a algorithm we should follow some guideline as follows:

a. Use plain language. In above example English is used. We can use our mother language also.
b. Do not use any programming language specific syntax. Same algorithm can be implemented in any programming language.
c. Every job to be done should be described clearly without any assumptions.
d. The developed algorithm must have a single entry and exit point.

An excellent algorithm should have the following properties.

Finiteness: Each algorithm should have finite number of steps to solve a problem.
Definiteness : The action of each step should be defined clearly without any ambiguity.

Inputs : Inputs of the algorithms should be defined precisely, which can be given initially or while the algorithm runs.

Outputs : Each algorithm must result in one or more outputs.

Effectiveness : It should be more effective among the different ways of solving the problem.

These properties should be satisfied by a good flowchart also.

2.2.1 Flowchart

A flowchart is a diagrammatic representation of an algorithm. It illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the design stage of computer solutions. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is essential for the better documentation of a complex program. Flowcharts facilitate communication between programmers and business people.

2.2.1.1 Symbols for flowcharting

Flowcharts are usually drawn using some standard symbols. Some of the frequently used symbols are discussed in table 2.1.

2.2.1.2 Basic Guidelines for drawing flowcharts

a. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
b. The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
c. The usual direction of the flow of a procedure or system is from left to right or top to bottom.
d. The number of entry and exit of flow lines in each symbol should be considered as mentioned in table 2.1.
e. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. We should avoid the intersection of flow lines to make it more effective and better way of communication.
f. Ensure that the flowchart has a logical start and stop.
g. It is useful to test the validity of the flowchart by passing through it with a simple test data.

Table 2.1 Brief description of flowcharting symbols

Symbol	Purpose	Description
→	Flow line	Used to connect symbols and indicates the flow of logic
(rounded rectangle)	Terminal (start / stop)	Used to indicate the start and end of a flowchart. One flow line exits from start and one enters to stop.
(rectangle)	Processing	Used whenever data is being manipulated, most often with arithmetic operations. A single flow line enters and a single flow line exits.
(parallelogram)	Input/Output	Used whenever information is entered into the flowchart or displayed from the flowchart. A single flow line enters and a single flow line exits.
(diamond)	Decision	Used to represent operations in which there are two possible alternatives i.e. decision making and branching. One flow line enters and two or three flow lines (labeled true and false) can exit.
(rectangle with double sides)	Predefined process/ Function	Used to represent a function call, or variables are sent to another function to return data or an answer. A single flow line enters and a single flow line exits.
(circle)	On-page Connector	Used to connect remote flowchart portions on the same page. One flow line enters or exits.
(pentagon)	Off-page connector	Used to connect remote flowchart portion on different pages. One flow line enters or exits.
- - - [	Comment	Used to add comments or clarification.
(cylinder)	Magnetic disk storage	Used to show the storage in magnetic disk. More than one flow line can enter and exit

2.2.1.3 Advantages of using flowcharts

Communication: Flowcharts are better way of communicating the logic of a system to all concerned.

Effective analysis: With the help of flowchart, problem can be analyzed in more effective way.

Proper documentation: Program flowcharts serve as a good program documentation, which is needed for various purposes.

Efficient Coding: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.

Proper Debugging: The flowchart helps in debugging process.

Efficient Program Maintenance: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

2.2.1.3 Limitations of using flowchart

Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.

Alterations and Modifications: If alterations are required the flowchart may require re-drawing completely.

2.2.3 Some examples to write algorithm and drawing flowcharts

Example 2.1 : Write algorithm and draw a flowchart to add two numbers.

Algorithm

Step 1. Start.
Step 2. Declare variables a, b, sum.
Step 3. Read values of a and b.
Step 4. Add a and b and assign the result to sum.
sum ← a + b
Step 5. Print sum.
Step 6. Stop.

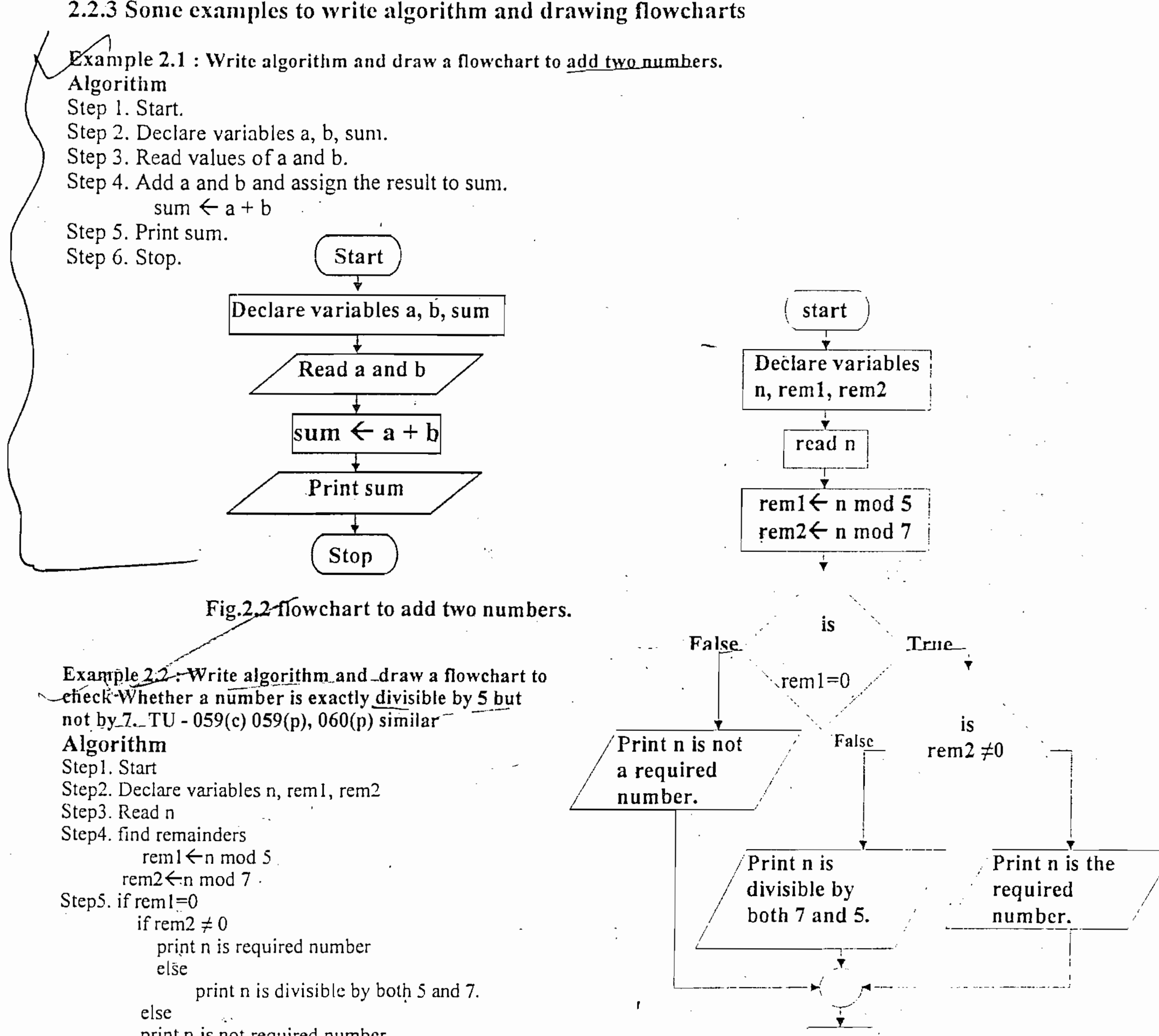


Fig.2.2 flowchart to add two numbers.

Example 2.2 : Write algorithm and draw a flowchart to check Whether a number is exactly divisible by 5 but not by 7. TU - 059(c) 059(p), 060(p) similar

Algorithm

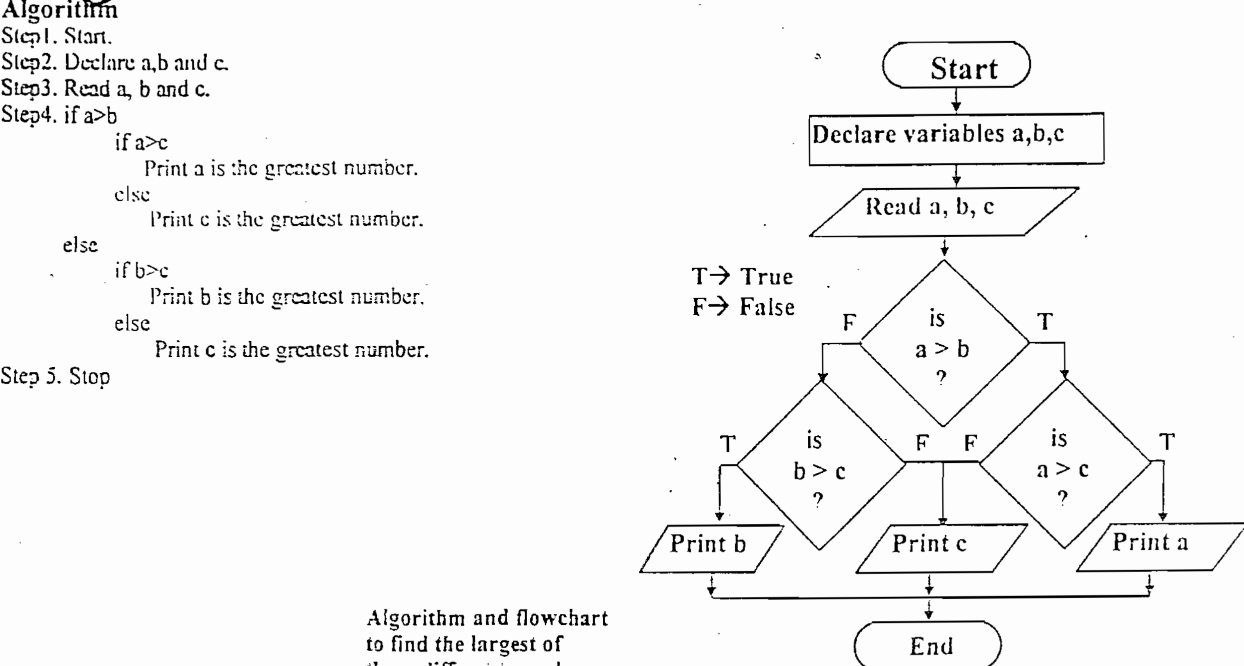
Step1. Start
Step2. Declare variables n, rem1, rem2
Step3. Read n
Step4. find remainders
rem1←n mod 5
rem2←n mod 7
Step5. if rem1=0
if rem2 ≠ 0
print n is required number
else
print n is divisible by both 5 and 7.
else
print n is not required number.
Step6. Stop

Fig. 2.3 Flowchart to check a number divisible by 5 but not by 7.

Example 2.3 : Write the algorithm and draw a flowchart to find the largest of three different numbers. TU- 058 chaitra

Algorithm

Step1. Start.
Step2. Declare a,b and c.
Step3. Read a, b and c.
Step4. if a>b
 if a>c
  Print a is the greatest number.
 else
  Print c is the greatest number.
else
 if b>c
  Print b is the greatest number.
 else
  Print c is the greatest number.
Step 5. Stop



Algorithm and flowchart to find the largest of three different numbers.

Example 2.4 : Write an algorithm and draw a flowchart to find all roots of a quadratic equation.

Algorithm

Step1. Start
Step2. Declare variables
 a,b,c,d,r,r1,r2,rp,ip
Step3. Calculate discreminant
 d←b²-4ac
 if d>0
  r1←(-b+√d)/2a
  r2←(-b-√d)/2a
  Print roots are real which are r1 and r2.
 else if d=0
  r ← -b/2a
  Print roots are equal which are equal to r.
 else
  Calculate real part(rp) and imaginary part(ip).
  rp←b/2a,ip←√ |d| / 2a
  r1←rp+j ip, r2←rp+j ip
  Print roots are imaginary which are r1,r2.
Step4. Stop.

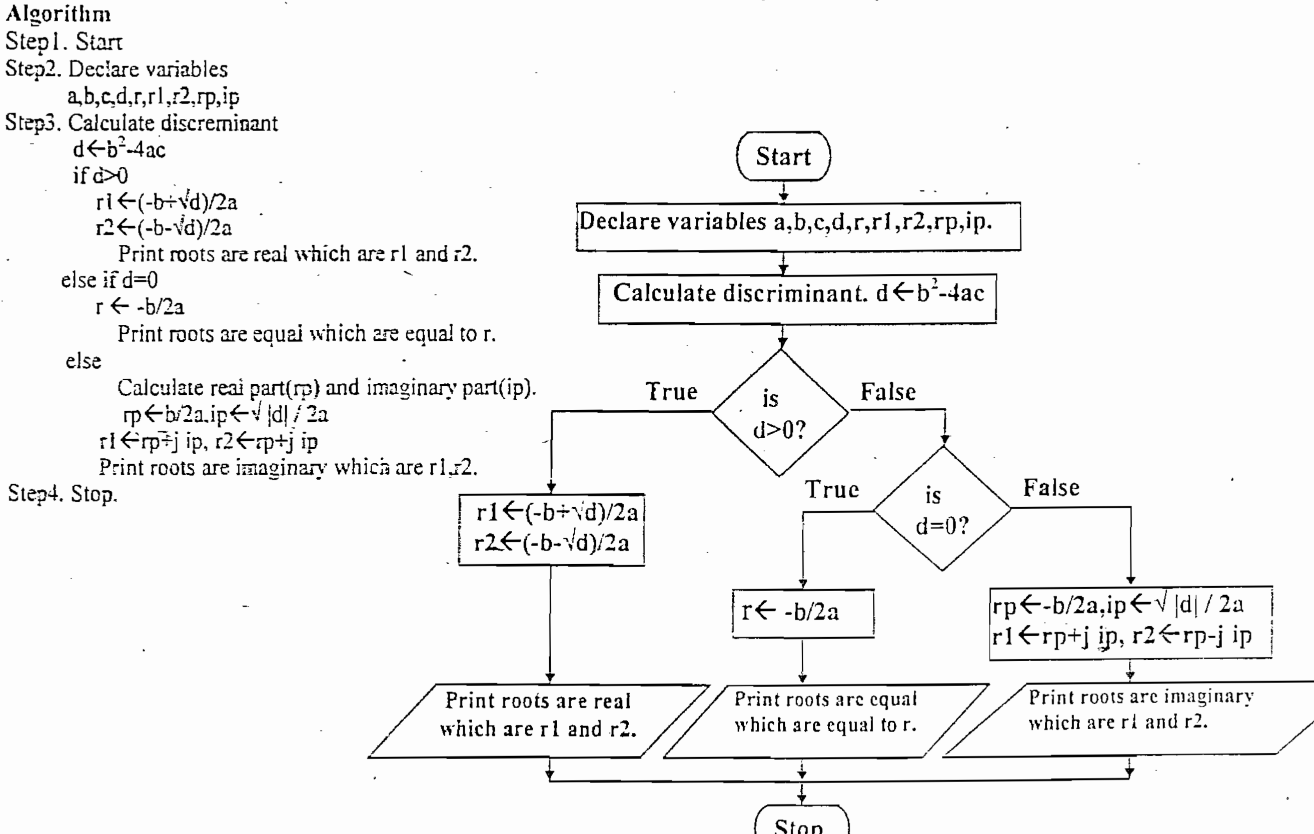


Fig.2.5 Flowchart to find all roots of a quadratic Equation.

Example 2.5 : Write algorithm and draw a flowchart for a program that calculates sum of digits of an integer number. TU - 2062 Baishakh

Algorithm

Step1. Start.
Step2. Declare variables n, rem and sum.
Step3. Initialize sum←0.
Step3. Read n.
Step4. Repeat steps 4.1,4.2, 4.3 while n ≠ 0.
    4.1 rem← n mod 10
    4.2 sum←sum + rem
    4.3 n← n /10
Step5. Print sum.
Step6. Stop

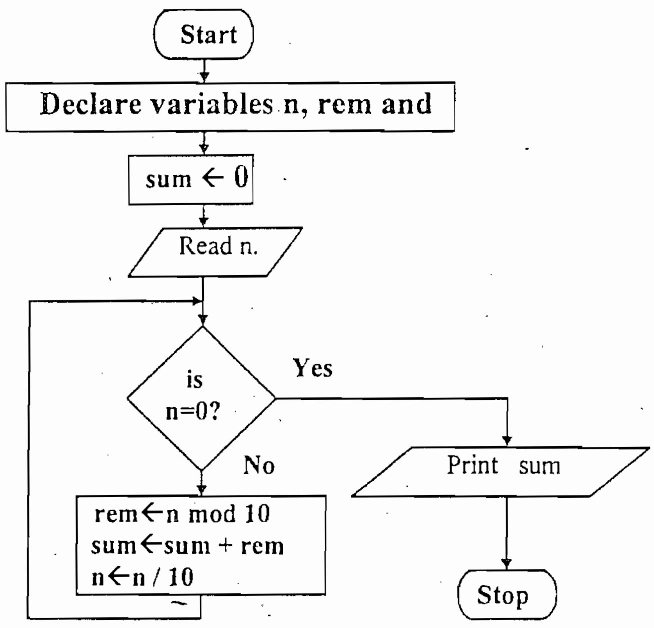


Fig. 2.6 : Algorithm and flow chart to calculate the sum of digits in an integer number.

Example 2.6 : Write algorithm and draw a flowchart to find factorial of a positive integer number entered by a user.

Algorithm

Step1. Declare variables n,i,fact.
Step2. Read n.
Step3. Initialize i←1, fact←1.
Step4. Repeat steps 4.1 and 4.2 while i≤ n
    4.1 fact←fact*i.
    4.2 i←i+1
Step5. Print fact.
Step6. Stop.

Flowchart

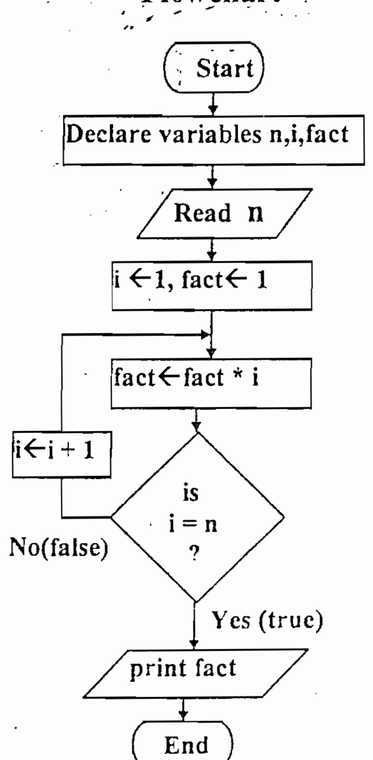


Fig.2.7 Flowchart to find factorial of a number entered by a number.

Example 2.7 : Write algorithm and draw a flowchart to find the HCF and LCM of two positive numbers.

Algorithm

Step1. Start
Step2. Declare variables a,b,c,d,rem,HCF,LCM.
Step3. Read a and b.
Step4. c←a, d←b.
Step5. rem ← c mod d
  if rem=0
    HCF←d
    Goto step 7.
  else
    c←d
    d←rem
    goto step 5.
Step7. Print HCF.
Step8. LCM←(a*b) / HCF
Step9. Print LCM.
Step10. Stop.

Start
Declare variables a,b,c,d,rem,HCF,LCM
Read a,b
c←a, d←b
rem←c mod d
is rem=0 ?
No
c←d
d←rem
Yes
HCF←d
Print HCF
LCM=(a*b)/HCF
Print LCM
Stop

Fig.2.8 flowchart to find HCF and LCM of two numbers entered by a user.

Example 2.8 : Write algorithm and draw a flowchart to find and print the terms of Fibonaci sequence till the term ≤500 and counting number of terms.

Algorithm

Step1. Start.
Step2. Declare variables
  fterm, sterm, term, nofterms.
Step3. Initialize variables.
  fterm←0,sterm←1,nofterms←2
Step4. term←ftermt+sterm
Step5. if term>500
    Print nofterms.
  else
    Print term.
    fterm←sterm
    sterm←term
    Goto step 4.
Step6. Stop.

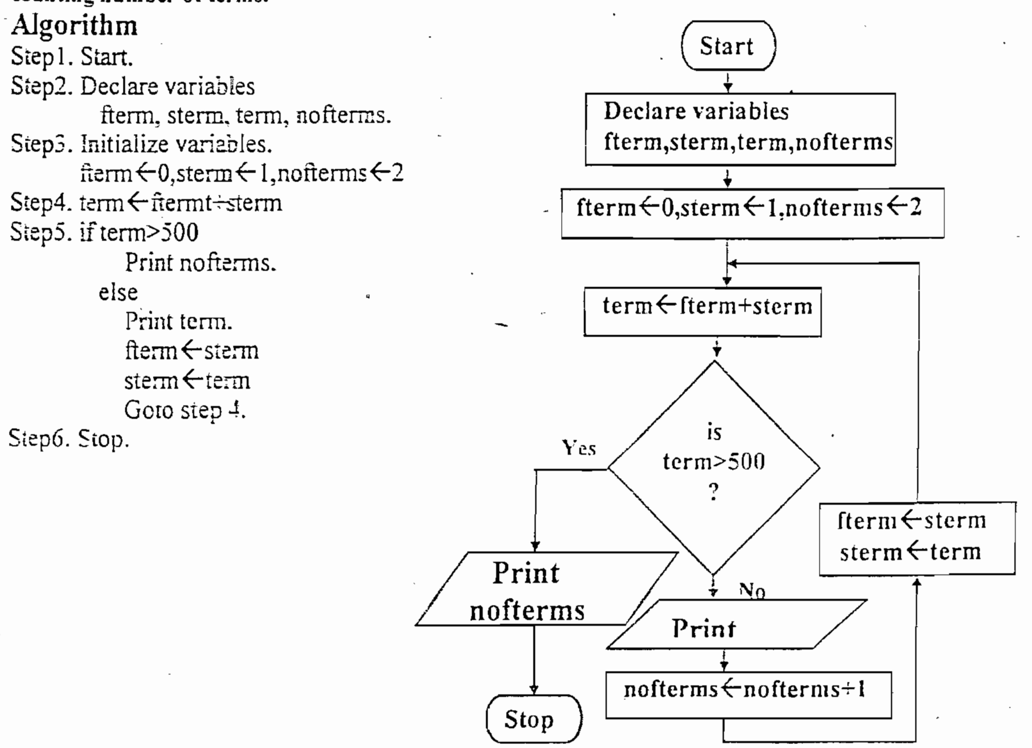


Fig.2.9 Flowchart to find terms of Fibonacci sequence.

Example 2.9 : Write an algorithm and draw a flowchart to check whether a entered number is prime number or not.

Algorithm

Step1. Start.
Step2. Declare variables i, rem and n.
Step3. Read n.
Step4. Initialize variables.
    i←2
Step5. rem← n mod i
    if rem=0
        Print n is not a prime number.
    else
        if i=n-1
            Print n is a prime number.
        else
            i←i+1
            goto step 5.
Step6.Stop.

Fig.2.10 Algorithm and flowchart check whether a entered number is prime number or not.

Example 2.10 Write algorithm and draw a flowchart to print multiplication table of nors by nocs.

Algorithm

Step1. Start.
Step2. Declare variables nors,nocs,i,j,product.
Step3. Initialize i to 1.
Step4.Read nors, nocs.
Step5. if i≤nors
        5.1 Initialise j to 1.
        5.2 if j≤nocs
            product←i*j
            Print product.
            j←j+1
            goto step 4.2
        else
            i←i+1
            goto step 4.
    else
        goto step 6.
Step6. Stop.

Flowchart

Fig.2.11 Algorithm and flowchart to print a multiplication table of order nors by nocs.

Example 2.11 : Write algorithm and draw a flowchart to find the largest element of an array age of size 10.

Algorithm

Step1: Start.
Step2: Declare variables age[10], i, max.
Step3: Initialize i. i←0
Step4: Repeat through step 5 till i remains less than 10.
Step5: Read age[i] , i←i+1
Step6: max ← age[0], i←1
Step7: Repeat through step 8 and 9 till i remains less than 10.
Step8: if age[i]>max
  max←age[i]
Step9: i←i+1
Step10. Print max.
Step11.Stop.

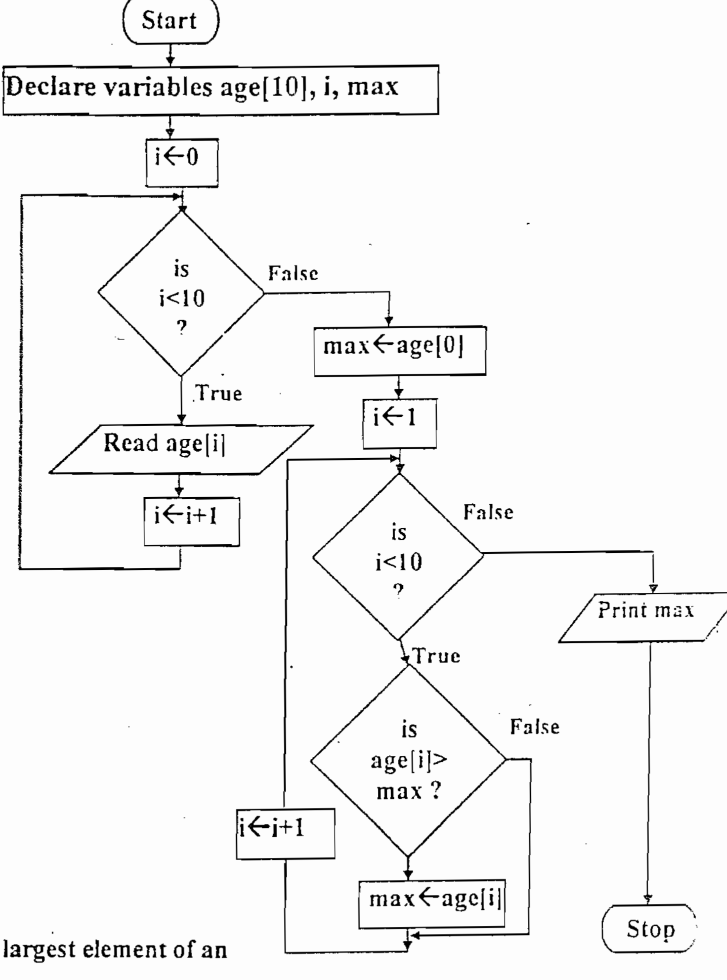


Fig. 2.12 Algorithm and draw a flowchart to find the largest element of an array age of size 10

Example 2.12 : Write algorithm and draw a flowchart to find smaller of two different numbers entered by a user using function.

Algorithm of calling function

Step1. Start.
Step2. Declare variables a,b,min and fuction smaller.
Step3. Read a,b.
Step4. Call function smallest with argument a and b.
  min←smaller(a,b)
Step5. Print min.
Step6.
End.

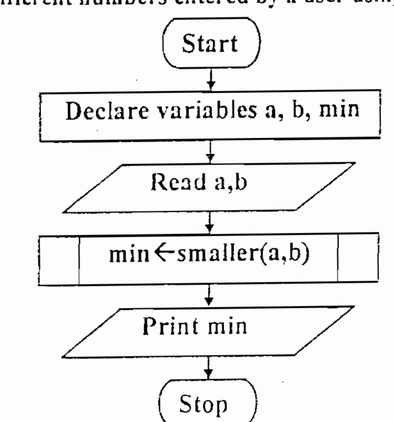


Fig. 2.13 Flowchart of calling function

Algorithm of called function (smaller)

Step1.Declare local variables x,y and smaller.
Step2. Assign a to x and b to y.
Step3. if x<y
  small←x
 else
  small←y
Step4. Return small.

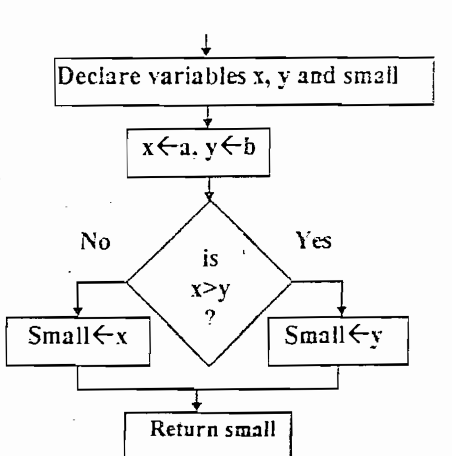


Fig. 2.14 Flowchart of called function

2.3 Coding

Above discussed algorithms cannot be read by the computers. It first must be written in a programming language to develop a computer program. Coding can be done in high or low level language. Now a days, many of the programmers use high level languages. Easily readable, reliable, proper error handling and easy to maintenance and support after installation are the features of an efficient program. The factors that makes program efficient are:

The variables can be carefully and discreetly named.

Comments can be inserted in to the program to document the logic pattern and program flow.

2.4 Compilation and execution

Program written in high-level language need to be converted to low level. Compiler does this task. Compiler, compilation process and execution are discussed in chapter 1.

2.5 Debugging and testing

Debugging and testing is the process of detecting and removing errors in a program, so that the program produces the desired results on all occasions. Before discussion of debugging and testing in detail, let us know types of errors.

2.5.1 Types of errors

Generally, errors are classified into following types.

2.5.1.1 Syntax error: Any violation of rules of the language results in syntax error.

2.5.1.2 Run-time Error: Errors such as mismatch of data types or referencing an out of range array element go undetected by the compiler. A program with these mistakes will run but produce the erroneous result.

2.5.1.3 Logical errors: These errors are related to the logic of the program execution. Such actions as taking the wrong path, failure to consider a particular condition and incorrect order of evaluation of statements belong to this category.

2.5.1.4 Latent errors: These are hidden errors that shows up only when a particular set of data is used.

Example:

r=(x+y)/(p-q)

This expression generates error when p=q.

2.5.2 Debugging

Debugging is the process of isolating and correcting any type of above discussed errors. Different types of debugging techniques are discussed below.

Error isolation

Error isolation is used for locating an error resulting in a diagnostic message. If we do not know the general location of the error, we can generally find the location of the error by temporarily deleting a certain portion of program and rerunning the program to see whether the error is again appeared or not. If the error is not appeared again, we know that the error is in the deleted code. If the error again appears, we know that the deleted portion is error free. Temporary deletion is accomplished by surrounding the instructions with comment markers(/* */). Similarly, we can put different unique printf statement in different parts (blocks) of the program to check whether that part(block) of the program is executed or not.

Tracing

In this technique, **printf** statement is used to print the values of some important variables at different stages of program. By observing these values, we can decide that whether the assigned values are correct or not. If any value is incorrect, we can easily fix the location of the error. Similarly, we can display and observe the values of variables that are calculated internally and assigned to the variables.

Watch values

A watch value is the value of a variable or an expression, which is displayed continuously as the program executes. Thus, we can see the changes in a watch value as they occur, in response to the program logic. By inspecting these values carefully, we can determine where the program begins to generate an incorrect or unexpected value. In Turbo C++ IDE(Interated Developent Environment), to define the watch values, go through the following steps

Debug→ Watchs→ Add watch ...→ specify variables or expression to watch its value→OK.

Breakpoints

A break point is a temporarily stopping point with in a program. Each breakpoint is associated with particular instruction with in the program. When the program is executed, the program execution will temporarily stop at the break point before the instruction is executed. The execution may be resumed until the next break point is encountered. Breakpoints are often used in conjunction with watch values, by observing the current watch value at each break points as the program executes. In Turbo C++ ,go through the following steps to set breakpoints.

Debug→Add→Breakpoint→provide the requested information in the dialog box.

Or

Select a particular line within the program→ press function F5.

To disable the breakpoint again press function key F5.

Stepping

The process of executing one instruction at a time is called execution. We can determine which instruction produce erroneous result or generate error messages by stepping through the entire program. Stepping is often used with watch

values, allowing us to trace the entire history of a program as it executes. Thus, we can observe changes to watch values as they happen. This allows us to determine which instructions generate erroneous results.
In Turbo C++, stepping can be done by pressing function key F7.

2.5.3 Testing

Software Testing is the process of executing a program or system with the intent of finding errors. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible. Typically, more than 50% percent of the development time is spent in testing. Testing is usually performed for the following purposes:

To improve quality.
For verification and validation.
For reliability estimation.

Testing process may include the following two stages:

2.5.3.1 Human testing

Human testing is an effective error detection process and is done before the computer based testing begins. Human testing method includes code inspection by the programmer and test group and review by a peer group. The test is carried out statement by statement and is analyzed with respect to a checklist of common programming errors. In addition to finding the errors, the programming style and choice of algorithm are also reviewed.

2.5.3.2 Computer based testing

Computer based testing involves two stages namely compiler testing and run time testing. We have studied about different types of debugging techniques, which are used to find the syntax error. If there is no syntax error, we need to find the logical and runtime errors. Which can be known after running the program. Run time error may produce the run time error message such as **null pointer assignment** and **stack overflow**. After removing these errors, it is required to run the program with test data to check whether the program is producing the correct result or not. Program testing can be done either at module (function) level or at program level. The module level test is the unit test. Where all the units are tested, they should be integrated to perform the desired function(s). An integration testing is done to find the errors associated with interfacing.

2.6 Program documentation

Program documentation starts from the starting of the software development life cycles. It keeps most of the information of all phases while developing projects. For example, in design phase we need to develop the algorithm and flowchart of the software being developed and these material should be documented properly for future reference. Similarly, there is a particular output in each phase for documentation. Documentation is used for future reference for both the original programmer and beginner. The final document should contain the following information:

- A program analysis document with objectives, inputs, outputs and processing procedures.
- Program design document algorithm and detailed flowchart and other appropriate diagrams.
- Program verification documents, with details of checking, testing and correction procedures along with the list of test data.
- Log is used to document future program revision and maintenance activity.

2.7 Assuring software quality

After all the above steps, we will have a software. We should be aware of the quality of the developed software. For assuring software quality, there are two techniques: technical review and structured work through. The principal objective of these techniques is to use participants' "fresh look" at the product (deliverable) to find as many errors, deficiencies, or problems as possible, so that they can be fixed before additional work is performed. This technique is based on the fact that the earlier an error is found, the cheaper it is to fix.

2.7.1 Technical Review

A Technical Review is an informal review by the project team of an intermediate deliverable (i.e. data model, function hierarchy, procedure logic, etc.). The product is for completeness, correctness, consistency, technical feasibility, efficiency, and adherence to established standards and guidelines by the Client.

2.7.2 Structured Walkthrough

The Structured Walkthrough is a review of the formal products produced by the project team (software development team). Participants of this review typically include end-users, software development team members, management of the client organization and development organization. As such, these reviews are more formal in nature with a predefined agenda, which may include presentations, overheads etc. The materials to be reviewed should be distributed to participants at least two days prior to the review date to ensure adequate time for review and preparation. Structured Walkthroughs are major milestones in the project and should be represented as such in the project plan and schedule. In structured walkthrough all the participants check a program's design to ensure it meets all requirements and is correct. Typically the program

to be checked is put through desk checking, in which the entire program is reviewed, that is the programmers check the printouts statement by statement, for different types of errors. Following this the program is further checked manually tracking the program, that is, by running sample data manually through the program. In structured walkthrough, documentation of all the software development phases are also reviewed.

Exercise 2

1. What is debugging and testing? How is it done? [2057/c/1-b]
2. What is a flowchart? List the various commonly used flowchart symbols. How does a flowchart help a computer programming? [2061/p/1-b]
3. What is algorithm development & flowcharting? What is its role in software development? [2058/p/1-b]
4. What an algorithm and flowchart to check whether a number is exactly divisible by 5 but not by 11. [059/c/1-b, 059/p/1-b, 2\060/p/1-b]
5. What are the steps that need to be followed for developing the application software? [2061/b/1-b]
6. Try to write to develop algorithm and flowchart for all the programming examples and programming problems the coming chapters.
7. Try to write algorithm and flowchart for all the programming examples of this book.
8. How many ways can we convert high-level languages to the machine-level language? Explain all in detail with block diagram. [2063/p/1]
9. Write algorithm and draw flowchart to print the following patterns:

a.
```
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1
```
[2063/b/2]

b.
```
5 4 3 2 1
5 4 3 2
5 4 3
5 4
5
```
[2063/p/2]

c.
```
A A A A B
A A A B B
A A B B B
A B B B B
B B B B B
```

Chapter 3
Introductory features of C
Operators and expressions

3.1 What is C ?

The C programming language was developed at Bell Laboratories of USA in 1972. It was designed and written by Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I(Programming Language-I), ALGOL(ALGorithmmic Oriented Language) etc. It was made the official Bell laboratories language. Thus, without any advertisement C's reputation spread and its pool of users grew. Possibly C seems so popular because it is reliable simple and easy to use. Initially, designed as a system programming language under UNIX it expanded to have wide usage on many different systems.

3.2 A historical development of C

By 1960 a crowd of computer language each for specific purpose, has come into existence. For example, FORTAN (FORmula TRAnslator) was being used for Engineering and Scientific Applications. COBOL was for commercial applications and so on. At that time, People had started to think rather learning a more general-purpose language than learning different special purpose languages. Therefore, an international committee was formed to develop such a language. This committee had developed a too general and too abstract language named ALGOL 60. To reduce this abstractness and generality a new language named Combined Programming Language (CPL) was developed at Cambridge University. It was not so popular because of its complexity to learn and implement. Basic Combined Programming Language (BCPL) was developed by Martin Richards at Cambridge University to solve the problems of CPL. But unfortunately it turned out to be too less powerful and too specific. Ken Tompson at Bell Laboratories had written a language called B around the time of BCPL. Like BCPL, it was also very specific. Ritchie inherited some good features of BCPL and B, added some his ideas and developed a new language called C. Lost generality in BCPL and B was restored and still keeping it powerful. As the language further developed and standardized, a version known as ANSI (American National Standards Institute) C became dominant. It still is used for some system and network programming as well as for embedded systems. More importantly, there is still a tremendous amount of legacy software still coded in this language and this software is still actively maintained. Each programming language has its own set of symbols to form alphabets, which are the smallest unit of a language. These can be used to form words. These words and other symbol are used for making a meaningful syntax. These syntaxes can be used to instruct the computer for data processing. A precise sequence of instructions is called a program. In this chapter, we will discuss the fundamental concepts of C programming language.

3.3 Character set

Characters are the fundamental units, which can be used to form words, numbers, constants, operators and expressions depending upon the computer on which the program is run. The characters available in C are categorized into following types:

a. **Letter**
b. **Digits**
c. **Special characters**
d. **White space**

a. letters
Uppercase A.........Z , Lowercase a..........z ~
b. Digits
All decimal digits 0....9
c. Special characters :Which are shown in table 3.1

Table 3.1 List of special characters in C

, comma	! exclamation mark	& ampsend	(left parenthesis
. period	/ slash	^ caret	) right parenthesis
; semicolon	\ back slash	* asteric	[left bracket
: colon	~ tilde	< opening angle bracket(less than)	] right bracket
? question mark	_ under score	> closing angle bracket (greater than)	{ left brace
' apostrophe	$ dollar sign		} right brace
" quotation mark	% percent sign		# number sign

d. White space characters - horizontal tab, blank space, new line, carriage return and form feed

3.4 Parts of a C syntax

If we observe any C syntax, we can not find other then the following tokens. In a passage of text, individual words and punctuation marks are called tokens.

3.4.1 Identifiers

What is your name? Suppose, your name is Ram. What does Ram indicates? It identifies you among the other persons around you. Why you have given this name? Is there no other such word? Of course, there is a large domain of such words. It is not compulsory to give the name Ram. Any other word can be used to identify. Similarly, in C every variables, functions, array, structure, unions etc should have their identification. Words used for this purpose are called identifiers. While creating identifier, uppercase and lowercase characters, digits, underscore (_) can be used. Identities must start with alphabetic character. The length of identifier should not be greater than 31 characters.

Table 3.2 C tokens

Type	Examples
Identifiers	main, a, add etc
Keywords	int, if,else etc
Integer constants	123, 1, 0, -100 etc
Real constants	12.3, -23.45 etc
Character constants	'a', 'b','4' etc
Strings constants	"my name is Ranjit Singh"
Operators	+, -,* , - etc
Special symbols	[],{}, &, ^ etc

3.4.2 Keywords

C keeps a small set of words for its own use. These are keywords. These keywords cannot be used as identifiers in the program. Each keyword has fixed meaning and that cannot be changed. They can not be used for any other propose. Keywords in ANSIC are listed in table 3.3. All key words must be written in lowercase.

3.4.3 Constants

What is the value of (3+4)? You say that 7.Yes, it is correct. But, can you say that the value of (3+x)? You will ask first for the value of x. Here, in the first case both operands of operator + have constant values. If you execute hundreds times, the answer is always 7 i.e., 3 and 4 remains same for each execution. But, in second case, if you execute hundred times there may be hundred different answers because there is a variable x. You can give the values of x differently in each execution. Similarly, In C Constants refer to fixed values that do not change during the execution of program. The C constants are shown in figure 3.1. Number associated constants are numeric constants and character associated constants are character constants.

3.4.3.1 Integer Constants

An integer constant refers to a sequence of digits. There are three types of integer constants : decimal, octal and hexadecimal. But here we only study decimal integer constants. Decimal integer consist of digits 0 through 9 preceded by an optional – or + sign.

Examples : 1,-40,+2000, 10000 etc.

Embedded space, comma and non- digit characters are not permitted between digits. For Example, 1,000, $2000 etc are not permitted.

For example, if we want to represent number of students we need to use integer constants

Table 3.3 Description of C keywords

Term	Description
auto	optional local declaration
break	used to exit loop and used to exit switch
case	choice in a switch
char	basic declaration of a type character
const	prefix declaration meaning variable can not be changed
continue	go to bottom of loop in for, while and do loops
default	optional last case of a switch
do	executable statement, do-while loop
double	basic declaration double precision floating point
else	executable statement, part of "if" structure
enum	basic declaration of enumeration type
extern	prefix declaration meaning variable is defined externally
float	basic declaration of floating point
for	executable statement, for loop
goto	jump within function to a label
if	executable statement
int	basic declaration of integer

Remaining part of Table 3.3

Term	Description
long	prefix declaration applying to many types
register	prefix declaration meaning keep variable in register
return	executable statement with or without a value
short	prefix declaration applying to many types
signed	prefix declaration applying to some types
sizeof	operator applying to variables and types, gives size in bytes
static	prefix declaration to make local variable static.
struct	declaration of a structure, like a record.
switch	executable statement for cases
typedef	creates a new type name for an existing type
union	declaration of variables that are in the same memory locations
unsigned	prefix declaration applying to some types
void	declaration of a type less variable
volatile	prefix declaration meaning the variable can be changed at any time
while	executable statement, while loop or do-while loop

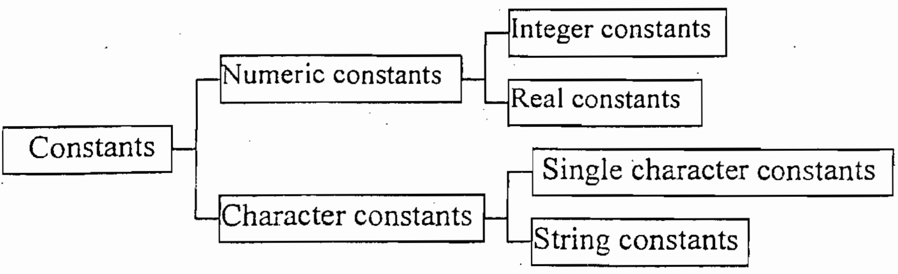


Fig. 3.1 Constants of C programming language

3.4.3.2 Real constants

If we want to represent any constants with fractional part, we must use real constants. For example, to represent ten and half we need to use write 10.5. Similariy, marks of students, interest rate in bank, daily atmospheric temperature, value of g in physics etc are real quantities. These real quantities vary continuously. In terms of C real constants are called floating-point constants. The real number also be expressed in exponential (or scientific) notation. For example the value 1234.5678 can be written as 1.2345678 e 3 in exponential notation. The general form is:

mantissa e exponent

The man[illegible] expressed in decimal notation or integer. The exponential is an integer number with an optional plus or minus sign. Exponential notation is useful for representing number that are either very large OS very small in magnitude For example, 4500000000 may be written as 4.5E10. Similarly, -0.000000123 is equivalent to –1.23 E-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. Some valid examples of numeric constants are given in table 3.4

Table 3.5 Character constants in C

Character type	No. of characters
Upper case letters	26
Lower case letters	26
Digits	10
Special symbols	32
Control characters	34
Graphic characters	128
Total	256

Table 3.4 Examples of numeric constants

Constants	Is it valid?	Why?
$123	No	$ symbol is not allowed
12,345	No	Comma is not allowed
1.2e 5	No	White space is not allowed
1.2e-3.4	No	Exponent must be an integer
12345L	Yes	Represents long integer
+2.0E6	Yes	ANSI C supports unary plus
4.5e-2,-4.5e-2	Yes	According to C rule

3.4.3.3 Single character constants

A single character constant (or character constant) contains a single character enclosed within a pair of single quotation marks. For example 'a' '4' ';' ' ', '*', '~' etc are character constants. Note that the character constants '4' and integer 4 are not the same. The fourth character is a blank space. When a character is typed into the computer via the keyboard, the character itself is not recorded. Rather, numeric version of that character is stored. Each character is assigned a number. This number is called ASCII value of that character. This is done to keep uniformity in communication. If characters are stored in computer as numbers how can one differentiate between a character and a number? It is required to maintain a conceptual separation by the programmer. We will discuss character data type to hold a single character. The size of character data type is 8 bit. The range of character is from –128 trough 127 in signed form and 0 through 255 in unsigned form. This is because 2^8=256. There are altogether 256 different characters used by IBM compatible family of computers (microprocessors). These characters are grouped as in table 3.5.

Out of 256 characters set the first 128 are often called ASCII characters and the next 128 as Extended ASCII characters. These characters are used for drawing single line and double line boxes in text mode. Each ASCII character has a unique appearance. The ASCII value of each character constants in decimal number system is shown in table 3.6.

How to read the table?

Observe that the character A is in row 6 and column 5. This means that the character A has ASCII value 65.

Some observations on the table

- Character codes 0 through 31 and 127 are control characters. Usually, they can not be printed by all versions of C.
- Character codes 32 and 255 print a single space.
- Character codes for digits 0 through 9, letters A through Z and letters a through z are contagious.
- Difference between a uppercase letter and the corresponding lowercase letter is 32. (lowercase=uppercase+32)

Table 3.6 ASCII values of character constants

	0	1	2	3	4	5	6	7	8	9
0	(NUL)	☺ (SOH)	☻ (STX)	♥ (ETX)	♦ (EOT)	♣ (ENQ)	♠ (ACK)	• (BEL)	◘ BS	HT
1	LF	♂ (VT)	♀ (FF)	♪ (CR)	♫ (SO)	☼ (SI)	► (DEL)	◄ (DC1)	↕ (DC2)	‼ (DC3)
2	¶ (DC4)	§ (NAK)	▬ (SYN)	↨ (ETB)	↑ (CAN)	↓ (EM)	→ (SUB)	← (ESC)	FS	GS
3	RS	US	SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^		`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	(DEL)	U	U
13	é	â	ä	à	å	ç	ê	ë	è	ï
14	î	ì	Ä	Å	E	æ	Æ	Ô	ö	O
15	û	ù	ÿ	Ö	ü	¢	£	¥	Pts	ƒ
16	à	í	ó	U	^	Ñ	ª	º	¿	─
17	¬	½	¼	¡	«	»	░	▒	▓	│
18	┤	╡	╢	╖	╕	╣	║	╗	╝	╜
19	╛	┐	└	┴	┬	├	─	┼	╞	╟
20	╚	╔	╩	╦	╠	═	╬	╧	╨	╤
21	╥	╙	╘	╒	╓	╫	╪	┘	┌	█
22	▄	▌	▐	▀	α	β	Γ	π	Σ	σ
23	µ	γ	φ	θ	Ω	δ	∞	Ø	ε	∩
24	≡	±	≥	≤	⌠	⌡	÷	≈	°	•
25	·	√	η	z	▪	(SP)				

Table 3.7 Meaning of some abbreviated ASCII symbols

Abbreviation	Meaning	Abbreviation	Meaning
NUL	Null character	SOH	Start of heading
STX	Start of text	ETX	End of text
ENQ	Enquiry	ACK	Acknowledgement
BEL	Bell	BS	Backspace
HT	Horizontal tab	LF	Line feed
VT	Vertical tab	FF	Form feed
CR	Carriage return	SO	Shift out
SI	Shift in	DEL	Data link escape
DC	Device control	NAK	Negative acknowledge
SYN	Synchronous idle	ETB	End of transmission block
CAN	Cancel	EM	End of medium
SUB	Substitute	ESC	Escape
FS	File separator (cursor right)	GS	Group separator (cursor left)
RS	Record separator (cursor up)	US	Unit separator (cursor down)
SP	Space(blank)	DEL	Delete

Escape sequence

Some non-printing and hard to print (backslash (\) and
apostrophe (')) can be expressed in terms of escape sequence. For example, the new line character (LF) is written as \n. Even though it is being described by two characters \ and n, it represents a single character. The backslash character \ is called the escape character and is used to escape the usual meaning of the character that follows it. The commonly used escape sequences are listed in table 3.8. Escape sequences can be written as '\a', '\n', '\"' etc.

Table 3.8 Escape sequences

Character	Escape sequence	ASCII value
Null character	\0	0
Alert (bell)	\a	7
Backspace	\b	8
Horizontal tab	\t	9
New line	\n	10
Vertical tab	\v	11
Form feed	\f	12
Carriage return	\r	13
Double quote	\"	34
Single quote	\'	39
Question mark	\?	63
Backslash	\\	92

3.4.3.4 String constants

A string constant is a sequence of any number of consecutive characters enclosed in double quotation marks. The characters may consist of letters, numbers, escape sequences and white spaces. Some examples of escape sequences are:

"Muskan Regmi" , "Kathmandu Engineering College", "01-4-284902" , "RS15,000" , "(a+b/e-3))", " ", "   ", "Line-

Note that " " is a empty(null) string.
The compiler automatically places a null character (\0) at the end of every string constants as the last character with in the string. This character is invisible with in the sting, which is used to identify the end of a string. We will study more about string in chapter 7.4(string).
Question : Explore the difference between 'a' and "a".

3.5 Variables

A variable is an identifier that is used to represent some specified type of information within a designated portion of the program. In other word, a variable is an identifier that is used to represent a single data item; i.e., a numerical quantity or character constants. The data item can be assigned in the program simply by referring to the variable name. A given variable can be assigned different data items at various places within the program. For example, a 250 CC cup is a variable and the tea in the cup is a data item. We can put other than the tea like water or coffee etc but characteristics of the cup remain the same. It means, the information (tea or water or coffee) represented by the variable can change but the data type associated (characteristics of the cup like size, color etc.) with the variable can not be changed during the execution of program.

Variable name can be chosen by the programmer in a meaningful way as to reflect its function or nature in the program. For example, if two values are to be added and their result stored, the variable containing the result might be called **sum**. Variable names may consist of letters, digits and underscore character. **sum, average, height, root_1, avg_weight, a, b** etc are some examples of variable names. While creating a variable name the following points should be keep in mind.

- They must begin with a letter. Some system permits underscore as the first character.
- ANSI standard recognizes the length of 31 characters. However, the length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.
- Uppercase and lowercase are significant. For example, SUM, sum and Sum are different variable names.
- Keywords can not be used as a variable name.
- Special characters e.g., white spaces, period, semicolon, comma, slash and other different types of operators are not permitted in variable names. (Note that the underscore is not considered as a special character in C).

123, &, (sqare_root), 100th, switch, Price$, first root are some invalid examples of variable names. Why they are invalid? Find yourself.
int_type is valid because keyword may be a part of a variable name.
average_weight is also valid because only first 8 characters are significant.

Question : Will there be any problem or not if there are two variables average_length and average_width when only the first eight characters are significant?

3.6 C data types

In computer science, a datatype is defined as a name or label for a set of values. These are used to defined the kind of data being stored or manipulated. In programming, the data type specifies the range of values that a variable or constant can hold and how that information is stored in the computer memory. C language is rich in its data type. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the need of the application as well as the machine.
ANSI C supports four classes of data types:

- Fundamental data type
- Derived data type
- User defined data type

3.6.1 Fundamental data type

All C compilers support four fundamental data types which are **character(char), integer(int), floating point(float)** and **double precision floating point(double)**. These are listed in table 3.9 with their keywords, memory requirement and range of values they can represent. These are discussed in the following topic with their alternation with qualifiers.

Table 3.9 Fundamental data types

Data type	Size	range
char	1	-128 to 127
int	2	-32,768 to 32,767
float	4	3.4e-38 to 3.4e+38
double	8	1.7e-308 to1.7e+308

3.6.1.1 Qualifiers

Qualifiers modify the behavior of the variable type to which they are applied. There are four types of qualifiers.

a. **Size qualifiers :** Size qualifiers alter the size of the basic data type. The keywords **long** and **short** are two size qualifiers. Both can be applied to **int**. Only long can be applied to double.
b. **Sign qualifiers :** Size qualifiers specify whether a variable can hold both negative and positive numbers or only positive numbers. The key words **signed** and **unsigned** are two sign qualifiers. These types of qualifiers can be applied to the data types int and char only.
c. const : This is new qualifier defined by ANSI standard. An object declared to be `const` can not be modified (assigned to, incremented or decremented) by a program. **const** variables can be declared as:

    const int p = 20;

but, the following segment of code is invalid.

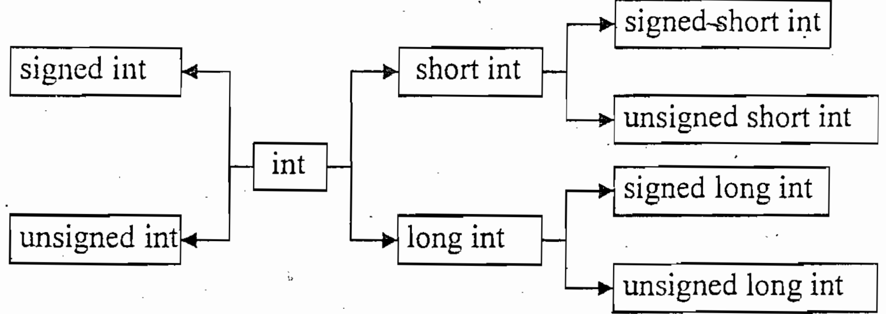
```
const int p = 20;
p = 10;      /* illegal */
```

d. **volatile** : This is new qualifier defined by ANSI standard. A variable should be declared **volatile** whenever its value can be changed by some external sources from outside the program. **volatile** variables can be declared as:

```
volatile int p = 20;
```

3.6.1.2 Integer data types

In mathematics, the natural numbers 0,1,2,3.....along with their negative numbers are integer numbers. These are whole numbers. On machine, only a finite portion of these numbers can be represented for integer type. Because the size of an **int** varies from one C system to another, the number of the distinct values that an **int** can hold is system dependent. Generally, integers occupy one word of storage, and since the word size of machine vary (typically, 16 or 32 bits) the size of an 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 the range -32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32-bit word length can store an integer ranging from -2, 147, 483, 648 to 2,147,483,647. In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int** and **long int**, in both **signed** and **unsigned** forms which is shown in figure 3.2 . For a regular **int** number uses, unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16-bit machine, the range of unsigned integer numbers will be form 0 to 65,535. We declare **long** and **unsigned** integers to increase the range of values. The uses of qualifier signed on integers are optional because the default declaration assumes a signed number. Table 3. 10 shows all the allowed combination of basic types and qualifiers and their size and range on a 16-bit machine.



3.6.1..**Fig. 3.2 Alteration of int data type by applying size and sign qualifiers**

A real data type consists of a subset of the real numbers. Floating point numbers in C are described by the keyword **float.** The possible values that a floating point type can be assigned are described in terms of attributes called range and precision. The precision describes the number of significant decimal places that a floating value carries. Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines) with 6 digits of precisions and range from 3.4e-38 to 3.4e+38. A real constant has the following form:

[sign][integer].[fraction][exponent]

where, sign→ an optional preceding sign, integer→ an optional integer part, .→a decimal point, fraction→ An optional fractional part, exponent→ an optional exponent part.

The integer and fraction part consist of one or more decimal digits. Either the integer or the fraction part may be omitted but not both. The exponent part consists of the letter 'e' or 'E' followed by an optionally signed integer constant of one or two digits. The exponent indicates that the value preceding it is to be multiplied by ten to the value of the exponent part's integer.

Table 3.10 Size and range of data types on 16 bit machine (Do not forget to see section 3.15.9)

Data type	Keyword	Size (in byte)	Range
Character or signed character	char or signed char	1	-128 to127
Unsigned character	unsigned char	1	0 to 255
Integer or signed integer	int or signed int	2	-32,768 to 32,767
Unsigned integer	unsigned int	2	0 to 65,535
Short integer or signed short integer	short int or signed short int	1	-128 to 127
Unsigned short integer	unsigned short int	1	0 to 255
Long integer or signed long integer	long int or signed long int	4	-2,147,483,648 to 2,147,483,647
Unsigned long integer	unsigned long int	4	0 to 4,294,967,295
Floating point	float	4	3.4e-38 to 3.4e+38
Double precision floating point	double	8	1.7e-308 to 1.7e+308
Extended double precision floating point	long double	10	3.4e-4932 to 1.1e+4932

3.6.1.4 Double precision real (double) data type

When the accuracy provided to a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits and range 1.7e-308 to 1.7e+308. Point to be noted is that the **double** represents the same data type that float represents but with a greater precision. To extend the precision further, we can use qualifier **long** to make it **long double** which uses 80 bits.

3.6.1.5 Character type

The character data type consists of ASCII character. A character value is stored in 8 bits(1 byte). Keyword **char** is used to define characters in C. A character variable can be **signed** (default) or **unsigned**. The signed **char** type has a range of –128 to 127 and **unsigned char** type has a range of 0 to 255;

3.6.1.6 Void type (see chapter 6)

The **void** is used to indicate that an expression has no value. No variables can be declared with such a type, but expressions may be cast to void. For example, the statement:

```
(void) printf("KEC");
```

Specifically indicates to the compiler that the return value from printf (an integer) is to be ignored. As such, the statement :

```
a = (void) printf("KEC");
```

is illegal because the assignment operator expects a value to be returned for assignment.

3.6.2 Derived data types

Functions, arrays and pointers are derived data types they are discussed in chapter 6 and 7.

3.6.3 User defined data types

C provides a feature to define new data types according to user's requirement. The ways of defining user defined data types are:

3.6.3.1 **Structure**-see chapter 8
3.6.3.2 **Union** – see chapter 8
3.6.3.3 **Enumeration** - see section 3.13

Hierarchy of C data types is shown in figure 3.3.

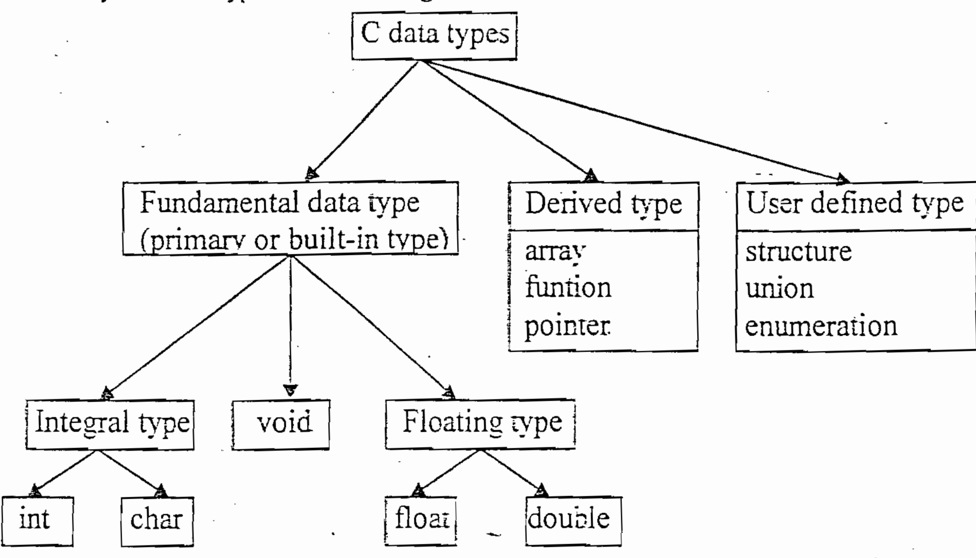


Fig. 3.3 Hierarchy of C data types

3.7 Declaration of variables

It is required to declare a variable to the compiler after deciding the variable name to use in a program. The declaration of variable must be done before they are used in the program. Declaration does two things:

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

The syntax for declaring a variable is as follows:

data_type variable_name

where **data_type** is one of the data type we discussed like int, float, unsigned long int etc or user defined data type and **variable_name** is any valid identifier.

For example,

```
int v1;
```

This means v1 is an integer type variable that can hold whole number within the range from –32,768 to 32,767. Many statements in C ends with a semicolon (;) like a sentence in English with a full stop. **int v1** is a statement and terminated by a semicolon (;). If there are more than one variables of the same type, they can be declared as

int v1,v2,v3,v4,v5..........vn;

3.8 Assigning values to variables

Before using any variables in the program for further processing, they must be given ..... two ways to assign values of variables. Which are:

- Using assignment statement
- Reading values of variables from keyboard using scanf library function

3.8.1 Using assignment statement

Values can be assigned to variables using the assignment operator (=)as follows:

variable_name=constant;

For example, in declarations:

```
int v1;
float v2;
char ch;
```

the values of v1, v2 and ch can be assigned as

```
v1=1245; [any value in the range -32,768 to 32,767]
v2=1234.345; [any value in the range 3.4e-38 to 3.4e+38]
ch='v'; [any ASCII character]
```

An assignment statement implies that the value of the variable on the left of the equal sign is set equal to the value of the quantity (or the expression) on the right. The statement

```
marks=marks+5;
```

means the new value of marks is equal to the old value of marks plus 5.

It is also possible to assign values to avariables at the time of its declaration. Its general form is as follows:

data_type variable_name=constant;

for example, above declarations and assignments can be done as

```
int v1=1245;
float v2=1234.345;
char ch='v';
```

This process of giving initial values to variables is called initialization. It is also possible to initialize more tham one variables in one statement using multiple assignment operators. For example, the statement a=b=c=d=e=0 is valid.

3.8.2 Reading values of variables from keyboard using scanf library function

Another way of giving values to variables is to input data through keyboard using the **scanf** function (see chapter 4 for detail). It is a general input function of C. The general form of **scanf** is as follows:

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

The control string contains the format of data being received. The address operator (&) before each variable specifies the variable name's address. For example,

```
scanf("%d",&v1);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable v1 to be typed in. **%d** in control string is the format specifier of integer data type so that we have to type in the value in integer form. **To assign the value to the variable, it is required to press the enter key.**

3.9 Structure of C programs

A function is a set of instructions designed to perform a specific task. Functions are building blocks of C programs. To create a C program we must create different functions and put them together. A C program normally includes sections shown in figure 3.4.

a. **Documentation section:** This section includes a set of comment lines giving the name of the program, name of the developer and other necessary detail of the program. Comments should be enclosed in /* */. Comments can be written anywhere in the program.
b. **Link section:** This section includes the instructions to the compiler to link functions from the system library. A large number of already defined general purpose functions are stored in system library
c. **Definition section:** This section defines all symbolic constants.(see more in 3.12)
d. **Global declaration section:** If any variable needs to be accessed by all the function in the program is declared in the global declaration section. (see more in chapter 6)

e. **main function section:** Every C program must have a **main** function to start the execution of the program. Body of main function consists of declaration and executable parts enclosed by the opening brace({) and the closing brace(}). The closing brace of the main function is the logical end of the program. The declaration part declares all the variables used in the executable part. There must be at least one statement in the executable part.

f. **Subprogram section:** The subprogram section contains all the user-defined functions that are called in the main function. These functions are generally placed immediately after the main function, although they may appear in any order.(see more in chapter 6)

All the sections except main functions are optional.

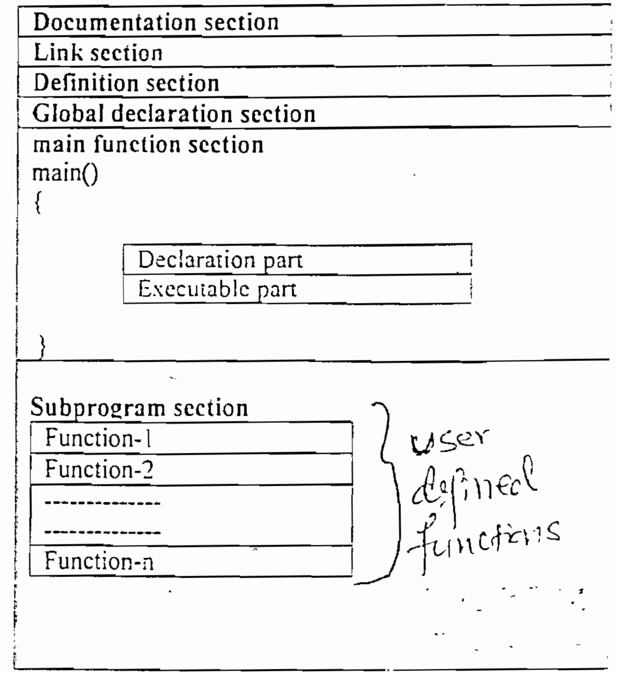


Fig 3.4 A general structure of C programs

Example 3.1 A hello world program.

```
1.  /* This is the first C program. It prints Hello
    World on the screen*/
2.  #include<stdio.h>
3.  void main(void)
4.  {
5.          printf("Hello world");
6.  }
```

Output

Hello world

Note: Line numbers are not parts of the program.

Description of program

Line 1: Comment line

It give some information about the program

Line 2: #include<stdio.h>

A preprocessor is built into the C compiler. When the command to compile a program is given, the code is first preprocessed and then compiled. Lines begin with a # communicate with the preprocessor. **stdio.h** is a header file. Which includes the declarations or more specifically the function prototypes (see chaper 6) of standard input/output factions of C system. The **#include** line causes the preprocessor to include a copy of header file **stdio.h** at this point in the code. The header file is provided by the C system. The angle bracket around **stdio.h** indicate that the file is placed in the usual location, which is system dependent. Here we have included this file to use standard output function **printf** because of the declaration of **printf** in **stdio.h.**

Line 3: void main()

It is the starting point of the program execution. main is function name. Each program must have a main function. A keyword **void** ahead of main tells the compiler that main function does not return any value. The parentheses following **main** indicate to the compiler that **main** is a function. void inside the parenthesis indicates that the main function do not take any arguments(see chapter 6).

Line 4: { (opening curly brace)

It indicates the beginning point of main function definition.

Line 5: printf("Hello world");

The C system contains a standard library of functions that can be used in programs. This is a function from the library that prints on the screen. We included the header file **stdio.h** because it provides certain information to the compiler about the function **printf()**. Here **Hello word** is a string constant which is given to the function **printf** as an argument (see chapter 6) to print it on the screen. It is ended with a semicolon (;) to indicate the end of statement.

Line 6: } (closing curly brace)

This indicates the logical end of the program. The right brace should match to the left brace. The set of statements enclosed with in a pair of curly braces ({ and }) is called a compound statement.

3.10 Running C programs in DOS

A C program must be converted into an executable file (i.e., a file with an exe extension) to run it in DOS. C++ is a superset of C so that C programs can be compiled by a C++ compiler. Compilers come in two forms which are in IDE(Integrated Development Environment) form and in command line form. The IDE includes all the facilities to develop and run programs, such as editor, compiler, debugger and so on. In case of Turbo C and Turbo C++, the IDE is a program called TC.EXE. Steps to create and run a C program on Turbo C (in IDE form) or Turbo C++ (in IDE form) compiler for DOS are listed below:

- Run TC.EXE file.
- Press Alt-F to open the file menu choose new command to open blank edit window.
- Write a program.
- To save, press F2 key and type the file name like firstprg.c in the text box and press OK.
- Press Alt-F9 to compile the program. If program is error free then press Ctrl-F9 to run the program.
- To see output press Alt-W and choose command **Output**.
- To exit from the TC editor press Alt-X.

Example 3.2 Write a program to add two numbers and display the result.

```
1.  #include<stdio.h>
2.  void main()
3.  {
4.      int sum, a, b; /* declaration section. There are three (a,b,sum) variables of integer type.*/
5.      a=5;          /*Giving value of a using assignment operator.*/
6.      b=10;         /* Giving value of b using assignment operator.*/
7.      sum=a+b; /*adding a and b and assigning the result to sum using assignment operator.*/
8.      printf("Sum of two numbers is %d", sum);  /* Printing value of sum using printf      function, %d   is
                      format that specifies(see chapter 4) for integer type data item.*/
9.  }
```

Output

Sum of two numbers is 15

Discussion of example 3.2 with respect to memory

Memory is storage space. It is divided into cells. Each cell is called a byte. Each byte comprises 8 bits. Each variable declared must be associated with sufficient amount of memory bytes. Each integer type occupies 2 bytes. It is required to reserve 6 bytes of memory to run this program. Each byte of memory is identified by a unique number called memory address. More about memory see chapter 7. Declaration of variables (a, b, sum) informs the compiler that a, b and sum are integer type variables and memory location must be reserved for them in memory. This concept is illustrated in figure 3.5. In declaration a two byte space starting from 1000 is reserved for variable sum. Similarly, from 1010 and 1022 for a and b. In line 5, value of a is given using assignment operator. This means that the given value 5 is stored at location 1010. Similarly in line 6, value of b i.e., 10 is stored at location 1022. The instruction of line 7 adds the content of location 1010(value of a) and 1022(value of b) and assigns to location 1000(address of variable sum). Line 8 prints the content of location 1000 i.e.,15 on the screen.

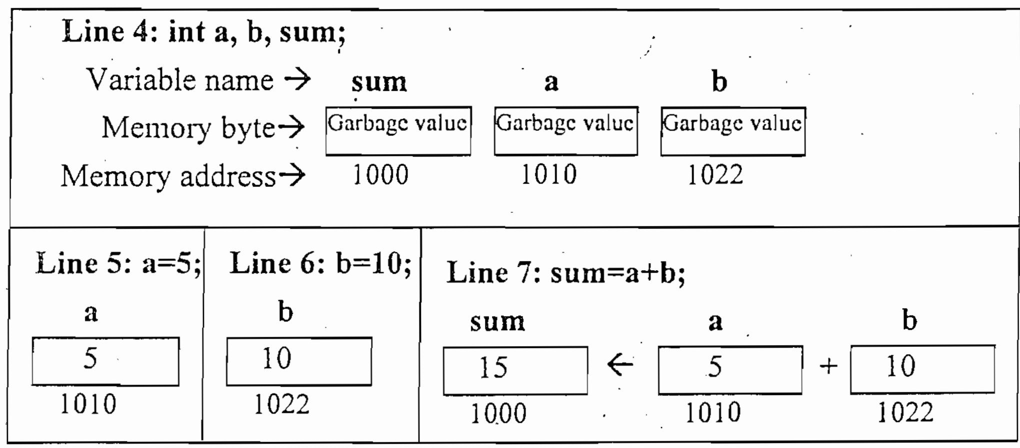


Fig. 3.5 Illustration of program execution with respect to memory

3.11 Library functions

C language is accompanied by an number of library functions that can carry out various commonly used operations or calculations. These library functions are not part of the C language. Some functions return a data item to their access point. Some indicate whether a condition is true or false by returning a 1 or a 0 respectively. Some carry out specific operations on data items but do not return anything. Library functions that are functionally similar are usually grouped together as object programs in separate library files. These library files are supplied as a part of each C compiler. All C compilers contain similar groups of library functions. Thus there may be some variations in the library functions that are available in different versions of the language. A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be

enclosed in parentheses and separated by commas. The arguments can be constants, variable names or expressions. The parentheses must be present even if there are no arguments. A function that returns a data item can appear anywhere within an expression, in place of constants or variables. A function that carries out operations on data item but does not return anything can be accessed simply by writing the function name. In order to use a library function it is required to include certain specific information (e.g. function declaration and symbolic constant definitions) within the main portion of the program. This information is stored in special files which are supplied with the compiler. Thus, required information can be obtained simply by accessing these special files. This is accomplished with the preprocessor statement

#include<filename>

Where filename represents the name of a special file. The commonly used file names are stdio.h, stdlib.h, math.h, string.h etc. The suffix "h" generally designates a "header" file which indicates that it is to be included at the beginning of the program. During the process of converting a C source program into an executable object program, the compiled source program may be linked with one or more library files to produce the final executable program. Some commonly used functions and their header files are in following section.

Note : The following function parameters are used.

c-character, d-double precision, f-file , i-integer, l-long integer, p-pointer, s-string, u-unsigned integer arguments and an asterisk(*) denotes a pointer.

a. <stdio.h> It contains declarations for input/output function.

Table 3.11 Functions in <stdio.h>

Functions	Return type	Purpose
fclose(f)	int	Close file f. return 0 if successfully closed
feof(f)	int	Determine if an end-of-file condition has been reached.If so, returns a nonzero value ; otherwise return 0.
fgetc(f)	int	Enter a single character form file f.
fopen(s1,s2)	FILE*	Open a file named s1 of type s2. Return a pointer to the file.
fprintf(f,...)	int	Writes data to the file f.
fputc(c,f)		Sends a single character to file f.
fputs(s,f)	int	Sends string s to file f.
fread(s,i1,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f to string s.
fscanf(f,...)	int	Enter data items from file f.
fseek(f,l,i)	int	Move the pointer for file f a distance l bytes from location i.
ftell(f)	long int	Return the current pointer position with in file.
fwrite(s,i1,i2,f)	int	Send s2 data items, each of size i1 bytes from string s to file f.
getc(f)	int	Enter a single character from file f.
getchar(void)	int	Enter a single character from the standard input device.
gets(s)	char*	Enter string s form the standard input device.
printf(...)	int	Send data item to the standard output device.
putc(c,f)	int	Send a single character to file f.
putchar(c)	int	Send a single character to the standard output device.
puts(s)	int	Send string s to the standard output device.
rewind(f)	void	Move the pointer to the beginning of file f.
scanf(...)	int	Enter data items from the standard input device.

b. <math.h> It contains declarations for certain mathematical functions

Table 3.12 functions in <math.h>

Functions	Return type	Purpose
acos(d)	double	Return the a arc cosine of d
asin(d)	double	Return the arc sin of d
atan(d)	double	Return arc tangent of d
atan2(d1,d2)	double	Return the arc tangent of d1/d2
ceil(d)	double	Return a value round up to the next higher integer.
cos(d)	double	Returns the cosine of d
cosh(d)	double	Returns the hyperbolic cosine of (d)
exp(d)	double	Raise e to the power d
fabs(d)	double	Return the absolute value of d
floor(d)	double	Return a value rounded down to the next lower integer.
fmod(d1,d2)	double	Return the remainder of d1/d2 (with same sign as d1)
labs(l)	long int	Returns the absolute value of l.
log(d)	double	Return the natural logarithm of d.
log10(d)	double	Returns the natural logarithm of d.
pow(d1,d2)	double	Return d1 raised to the d2 power
sin(d)	double	Return the sin of d
sinh(d)	double	Return the hyperbolic sine of d.
sqrt(d)	double	Returns the square root of d
tan(d)	double	Return the tangent of d
tanh(d)	double	Return the hyperbolic tangent of d.

c. <string.h> It contains declarations for some string handling functions.

Some important function in this header files are discussed in chapter 7.4(string).

d. <ctype.h> It contains declarations for character handling and conversion functions.

Some important function in this header files are discussed in chapter 7.4(string).

e. <stdlib.h> It contains declarations for utility functions such as string conversion routines, memory allocation routines, random number generator etc.

some string conversion functions in this header files are discussed in chapter 7.4(string) and some memory management functions are discussed in chapter 7.3(Dynamic Memory Allocation).

These are only the frequently used header files and functions but there are many other header files and functions. We can see them in the help of any C or C++ compiler. We can search the functions and use them as per our requirements.

3.12 Defining Symbolic Constants

Sometime we need to use some constants in a program repeatedly. One example of such a constants is π=3.1415.

If we use 3.1415 in many places in our program, then we will have manly two problems:

- To modify the program
- To read and understand the program.

To solve these problems, we use #define preprocessor directive. Its general form is as follows:

#define symbolic-name values of constant

Where #define is a preprocessor directive. It falls under definition section of a C program. Symbolic-name is a constant. Sometimes, it is called a constant identifier. For example,

#define PI 3.1415

Where PI is a symbolic-name and 3.1415 is the value of PI

Example 3.3 This program calculates area and circumference of a circle. The program commands the user to enter the radius of circle.

```
1.  #include <stdio.h>
2.  #define PI 3.1415   /* using define directive */
3.  void main()
4.  {
5.        float area, radius, circumf;  /* variable declaration */
6.        printf("Enter radius of circle:");
7.        scanf("%f",&radius);  /* using scanf function to read values of variable radius from keyboard */
8.        area=radius*radius*PI;  /* Calculation of area using mathematical formula */
9.        circumf=2*PI*radius;   /* calculation of circumference */
10.       printf("\nArea of the circle= %f", area);
11.       prinf("\nCircumference of the circle is= %f", circumf);
12. }
```

Output

```
Enter radius of circle: 12.5
Area of the circle= 159.391495
Circumference of the circle is= 78.537498
```

Line2: value of symbolic constant PI is defined its value is 3.1415.

Line7: **scanf** function reads the value of variable radius from the keyboard(standard input device) and stores the value at address of radius using address operator(&).

Line 8,9: Symbolic constant PI is used instead of value of π (3.1415). To modify the value of π (3.1415), we need not do more than changing 3.1415 to the required value in include directive. It would be very difficult to modify and understand, if 3.1415 was written instead of PI.

Line 10: **printf** function is used to print the content of variable **area.** Format specifier **%f** is used for float type variables. **The following rules apply to a #define statement which define a symbolic constant.**

- Symbolic names are similar to variable names. They are generally in UPPER CASE.(not compulsory)
- Blank space between # and **define** is not allowed.
- A blank space is required between **#define** and **symbolic-name** and between **symbolic-name** and the **constant.**
- Symbolic names are not declared for data types. Its data type depends on the type of constants.

More about preprocessor compiler directive (#define) is discussed in chapter 10.

3.13 Enumeration [It is better to study this section after studying chapter 8]

Enumeration is a technique that defines a set of integer type constants. In C, an enumerator is of type **int**. But the enumeration is useful to define a set of constants instead of using multiple #defines. Enumeration is a way of creating of user defined data type. Keyword **enum** is used to create enumerated data type. The general form of creating enumerated data type is as follows:

enum identifier{ value1, value2, value3,……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 with in the brace (enumeration constantans). After this definition, we can declare variables to be of this new type as below:

enum identifier variable1, variable2…variablen;

The enumerated variables variable1, variable2…variablen can only have the values value1, value2, value3,……valuen. The assignment of following type is valid:

variable1=value1;
variable2=value2;

Sometime we need to define list of symbolic constants, which represent integer numbers. For example,

```
#define SUN     1
#define MON     2
........................
```

```
........................
#define SAT          7
```

The set of constants can be defined using **enum** as follows:

Example 3.4 Illustration of enumerated data type

```
#include<stdio.h>
void main(void)
{
        enum days {SUN=1, MON, TUE, WED, THU, FRI, SAT};
        enum days d, day1, day2, day0, day10, d1; /* Declaration of variables of type days */
        d=MON;
        printf("%d",d);
        day10=SAT;
        if(day10= =SAT)
        printf("It is last day of the week.");
        printf("%d",SUN+MON);
}
```

Output

2 It is last day of the week. 3

Here value of SUN is 1 which is explicitly assigned. The value of MON is 2, TUE is 3 and so on i.e., incrementing by 1. If it is not given compiler assigns 0 to SUN. Above example shows that enumeration variables can be processed in the same manner as the integer variables. Thus, they can be assigned new values, compared etc. Enumeration variables are generally used internally to indicate various conditions that can arise within a program. An enumeration constant can not be read into the computer and assigned to an enumeration variable. Look at the following examples to understand how to assign values to the constants in enumerations.

enum Color { RED, BLUE, GREEN};

Unless specified, the first constant has a zero value. The values increase by one for each additional constant in the enumeration. So, RED equals 0, BLUE equals 1, and GREEN = 2. The values of each constant can also be specified. In syntax,

enum SHAPE {SQUARE= -5, RECTANGLE, TRIANGLE=17, CIRCLE, ELLIPSE};

The value of SQUARE = -5, RECTANGLE= - 4, TRIANGLE=17, CIRCLE =18 and ELLIPSE=19.

3. 14 User defined data type name

C allows the definition of our own types based on other existing data types. We can do this using the keyword typedef, its format is:

```
typedef  existing_type  new_type_name ;
```

where existing_type is a C data type and **new_type_name** is the name for the new type we are defining. Actually it is the technique to rename the C data type according to own interest. **typedef** does not create different types. It only creates synonyms of existing types.

Eexample 3.5 Illustration of typedef.

```
#include<stdio.h>
void main()
{
        typedef  float marks;
        marks m1,m2,average;
        printf("Enter value of m1 and m2:");
        scanf("%f%f",&m1,&m2);
        average=(m1+m2)/2;
        printf("Average=%f",average);
}
```

In line 4 of example 3.5, fundamental data type **float** is renamed to **marks**. In line 5, variables m1,m2 and average are declared for marks type and in line 7, values of variables m1 and m2 are read from the keyboard using a single scanf function.

Uses of typedef

- to define an alias for a type that is frequently used within a program.
- if a type that we want to use has a name is too long or confusing.

3.15 Operators and expressions

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expression. For example, +,-,>,%, sizeof are some of operators. The data items that operators act upon are called **operands**. On the basis of operands, they take to act upon, operators are classified into unary, binary operators and ternary. The operators that require two operands are called binary operators.

For example,

a + b

In this expression, a and b are two operands that are used by addition operator (+) to perform its task.

Similarly, the operators that require one operand to perform its task are called unary operators.

For example.

- 5

where – is a unary operator that converts positive 5 to negative 5. Some other unary operators are +, ++, --, **sizeof** and (type) are unary operators. These are discussed in the coming sections.
For ternary operator see section 5.2.3.

An expression, in general, is a combination of variables, constants and operators written according to the syntax of the language. In C, every expression results in some value of a certain type that can be assigned to a variable. Single data items such as a number, a character or variables are also considered as expressions. Expression can also represent logical conditions that are either true or false. In C, true is means a nonzero value (considered as 1) and false means zero (1). For example, a, b and a+b are expressions in **a+b**. If a=5 and b=10, then expression a+b yields 15.
Now, what will be the value of
a>b
It yields false(0) because a(5) is less then b(10).
Based on their utility and actions, C operators are classified as follows:

- Arithmetic operators
- Relational Operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Special operators

3.15.1 Arithmetic operators

The arithmetic operators perform arithmetic operators. These can operates on any built-in data type allowed in C. There are both unary and binary arithmetic operators in C. Which are shown in table 3.13. The unary minus operator multiplies it single operand by –1.
The operands acted upon by arithmetic operators must represent numeric values. The remainder (%) operator requires both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operands be nonzero. C does not have an operators for exponentiation. For this purpose **pow** function in math.h can be used. Arithmetic operations in C are classified in to three types.

Table 3.13 Arithmetic operators

Operators	Meaning
+	Unary plus
-	Unary minus
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder(modulo division)

- Integer arithmetic
- Real arithmetic
- Mixed mode arithmetic

3.15.1.1 Integer arithmetic

An arithmetic operation involving only integer operands is called an integer arithmetic. The most important point to know here is that **integer arithmetic yields always an integer value.**
For example, if **int x=5, y=17;**
Then, -(5)=-5
x + y=22
x - y= - 12
x * y= 85
y / x=3 [fractional part is truncated in integer part]
x /y = 0
y % x=2[remainder]
x + 'A' = 70 [ASCII value of A is 65]
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 truncation is implementation dependent. That is 4/5 =0 and –4/-5 =0 but –4/5 may be 0 or –1(machine dependent). In remainder the sign of the result is always the sign of the first operand (dividend). –10%3= -1, -10%-3= -1, 10%-3= 1 are some examples.

3.15.1.2 Real arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume value either in decimal or exponential notation. The floating points results are allowed to round off to the number of significant digits specified and hence the final value is only an approximation of the correct result. The remainder operator is not applicable to real operands. For example, if
float a=10.5, b= -5.7;
then
a+b = 4.8, a-b =16.2, a*c = - 59.89, b/a = -0.542857

3.15.1.3 Mixed mode arithmetic

In mixed mode arithmetic, if either of the operands is real then the resultant value is always a real value. The following examples show the concept clearly.
if **float x = 17.0; int y = 5;**
then 17.0 / 5 = 3.4000000, 5 / 17.0 = 0.294118

int a=90, b = 50, c;
c= a + b;
here, I. The values of a+b are evaluated first.
II. The result (a+b) is assigned to c.

There are other arithmetic operators which are known as shorthand arithmetic operators. The general form is

v op= exp; [equivalent to v = v op exp in normal case]

Where v is a variable, op is a C binary operator and exp is an expression. For example, x − = 5; is same as the x = x − 5. A list of assignment operators is shown in table 3.19. Some equivalent expressions are given in the table 3.20. Easy to read and write programs with more concise and efficient coding are benefits of using shorthand operators in programming.

Table 3.19 list of assignment operators

Operators	Meaning
=	Simple assignment
*=	Assign product
/=	Assign quotient
%=	Assign remainder (modulus)
+=	Assign sum
-=	Assign difference
&=	Assign bitwise AND
^=	Assign bitwise XOR
|=	Assign bitwise OR
<<=	Assign left shift
>>=	Assign right shift

3.15.5 Increment and decrement operators

++ and -- are increment and decrement operators. Both are unary operators. ++ adds 1 to the operand and -- subtracts 1 from the operator. They are extensively used in loop statements (see chapter 5). The concept of increment operator is shown in table 3.21.

Table 3.20 some equivalent expressions

Statement with short shorthand operator	Statement with simple assignment operator
x -= 5	x = x - 5
y += 10 * p	y = y + 10 * p
z *= 15	z = z * 15
x / = (a+20)	x = x / (a+20)

Table 3.21 Illustration of increment operators

Operator type	General form	Meaning	Example (if a=5)	Equivalent expression	Result Final value of a
Prefix	++ <variable>	Pre increment	++a	a=a+1	6
Postfix	<variable> ++	Post increment	a++	a=a+1	6

Above examples shows that in both case final result is same. While ++a and a++ mean the same thing when they form statement independently, they behave differently when they are use in expressions on the right hand side of assignment statement. For example,

If p = 10;
q = p++;

Steps to execute the second statement (q=p++).
Step 1. The value of p will be assigned to q i.e., latest value of q is 10.
Step 2. The value of p will be incremented by 1 and assigned to p i.e., latest value of p is 11.
Theme is assigning before incrementing.
Again if p = 10;
q = ++p;
Steps to execute the second statement (q=++p).
Step1. The value of p is incremented by 1 and assigned to p i.e., latest value of p is 11.
Step2. The value of p is assigned to q i.e., the latest value of p (11) is assigned to q. It shows that the latest value of q is also 11.
Theme is incrementing before assigning.
Similarly, we can understand decrement operators (prefix (--p) and postfix (p--)).

3.15.6 Conditional operator

The conditional operator consists of two symbols: question mark (?) and colon (:). It takes three operands so it is ternary operator. More about this operator see section 5.2.3.

3.15.7 Bitwise operators

A bitwise operator operates on each bit of data. These are required in bit level programming. These are used for testing, complementing or shifting bits to the right or left. These are not applicable to real data (float and double). A list of bitwise operators is shown in figure 3.22. Discussion in detail is out of the scope of this text.

Table 3.22 A list of bitwise operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right
~	One's (1's) complement

3.15.3 Comma operator

C has a feature to link a chain of statements together using a comma operator to separate them. Some of the usages of comma operators are:

a. In variable declaration:
For example p, q, r and s can be declared by the statement
int p, q, r, s;
b. In the following type of statements:
p=(q=50, r=-20, s=10, q + r + s);
comma linked list of expression are evaluated from left to right and the value of the rightmost expression is [illegible] above statement first 50 is assigned to q and then −10 is assigned to [illegible] 10. This result is assigned to the p.

c. In loops (see chapter 5)
d. To swap values: e.g., p = q, q = r , r = t;

3.15.9 The sizeof operator

It is a unary operator. It takes argument as a constant, a variable, a data type qualifier or a more complex expression and returns the size of that operand on the current compiler in bytes. The **sizeof** operator is very useful in developing portable programs. It is a bad practice to assume the size of a particular type since its size may vary from compiler to compiler. So, using **sizeof** operator to know the size of any data type, constant, expression, array or structure is the best practice. The use of size of operator is more illustrated in chapter 7.3(Dynamic Memory Allocation).

Example 3.6 : size of illustration of sizeof opertor.

```
#include<stdio.h>
void main(void)
{
        int a=9;
        float b=4.6;
        clrscr();
        printf("\nsize of 5 is=%d",sizeof(5));
        printf("\nsize of a =%d",sizeof(a));
        printf("\nsize of a+b =%d",sizeof(a+b));
        printf("\nsize of long int =%d",sizeof(long double));
}
```

Output

```
size of 5 is=2
sezeof a =2
sezeof a+b =4
sezeof long int =10
```

3.15.10 Pointer operators (* and &, see chapter 7.2.3)

3.15.11 Member selection operators (. and →, see chapter 8)

3.15.12 Precedence and associativity of operates

Consider the following statements:

int x=20, y=30, z=40, reslt ;

reslt = x - y * z ;

In the above expression, there are two operators: - and * . Which operator operates first? x-y or y*z ? We have already known that both – and * are arithmetic operators. We have already known that * has higher priority then the -. Now, we can easily say that y is multiplied by z first. The product is subtracted from x and the final result is assigned to variable reslt. This priority of operators to operate in an expression having different level of operators is called operator precedence.

Now see the next statement

reslt=x – y + z ;

Again we have the similar question: whether y is subtracted from x first or y is added to z. Which operand operates first – or +?. Here both operands – and + have the same priority. So this case is more confusing than the previous one. In such condition C allows the operators to operate according to left to right order or right to left. In case of arithmetic operators, it is left to right. Now, we can say that – operator operates first because it is the left most operator in the expression. Then + operator gets its turn to operate. This ordering of operation of operators of same priority level in a expression is called operator associativity.

Parentheses allow us to change the order of priority because of their highest precedence. When in doubt, we can add an extra pair just to make that the priority assumed is the one we require. Parenthesis can also be used to improve readability of the program.

Question: Which operator in the following expression gets the first chance to operate?

reslt=(x-y)*z;

Above example shows the concept of arithmetic operators. In C, we have discussed different types operators. Each operator in C has their own precedence and associativity. The overall summary of C operators with their precedence and associativity is shown in table 20.23.

Example 3.7: Illustration of evaluation of an expression

let us assume an expression

f=(a + b) - (d-c) / e + (d ÷ e) * c

Analysis of the expression

Question: How many types of operators are there?

Answer: Arithmetic, function call (parenthesis) and assignment which are: (), +, -, / ,* and +.

Question: What is the priority of each operator?

Answer: ( ) have 1, / and * have 3, + and – have 4 and = has 14.

Question: What is the associativity of the operators?

Answer: Assignment operator (=) right to left and all other has left to right.

When a=10,b=12,c=15,d= 30 and e=5, the statement becomes

f = (10 + 12) – (30 – 15) / 5 + (30 ÷ 5) * 15

and is evaluated as follows:

First pass	Step1: f=10-(30-15)/5+(30+5)*15 Step2: f=10-15/5+(30+5)*15 Step3: f=10-15/5+35*15	Evaluation is started form the right side of assignment operator. On the right side, there are three sets of parenthesis. Due to the highest precedence and associativity from left to right, the evaluation is started from the expression contained in the left most set of parentheses and end to the rightmost parentheses. Finally, the expression has only arithmetic operators.
Second pass	Step1: f=10-3+35*15 Step2: f=10-3+525	In second pass, division and multiplication have same precedence. so according to left to right associativity division is done before multiplication.
Third pass	Step1: f=7+525 Step2: f=532	In third pass, subtraction and addition have the same precedence but associativity is left to right so subtraction is done before addition. Finally the result is assigned to f.

Example 3. 8 : If a=10,b=20,c=5,d=15, what does the following expression yields?
(a== (c –2) && c- ((d + 10) < 25)|| c!=b)

Solution, →(10==(5-2) && ((15+10)<25) || 5!=20)
→(10 == 3 && ((15+10)<25||5!=20)
→(10==3 && (25<25||5!=20))
→(10==3&& (false || 5!=20))
→(10==3&& (false || true))
→(10==3&&true)
→(false && true)
→false

3.15.13 More on arithmetic expressions

How to write C expression form algebraic expression ? see the following examples.

1. a x b = a * b
2. $a^2 + 2ab + b^2$ = a * a + 2 * a * b + b * b
3. (ab)/b = (a * b) / b
4. (x+y)(x-y) = (x + y) * (x – y)

and so on. Blank space around an operator is optional and adds readability. Before using any variables in the statements, they must be declared.

15. 313.1 Evaluation of expression

Arithmetic expressions are evaluated using assignment statement. Which is already discussed.

Table 20.23 Summary of C operators with their precedence and associatvity

Operator	Operation	Precedence	Associativity
() [] -> .	Functional call Array element reference Indirect member selectio Direct member selection	1	Left to right
! ~ + - ++ -- & * sizeof (type)	Logical negation Bitwise(1's) coplemtnt Unary plus Unary mius Preincrement or postincrement Predecrement or postdecrement Address Pointer reference(indirection) Returns the size of an object in bytes Type cast (conversion)	2	Right to left
* / %	Multiply Divide Remainder(odulus)	3	Left to right
+ -	Binary plus(addition) Binary minus(subtraction)	4	Left to right
<< >>	Left shift Right shift	5	Left to right
< <= > >=	Less than Less than or equal Greater than Greater than or equal	6	Left to right
= = !=	Equal to Not equal to	7	Left to right
&	Bitwise AND	8	Left to right
^	Bitwise exclusive XOR	9	Left to right
|	Bitwise OR	10	Left to right
&&	Logical AND	11	Left to right
||	Logical OR	12	Left to right
?:	a ? x : y means "if a then x, else y".	13	Left to right
= * = / = % = + = - = & = ^ = | = <<= >>=	Simple assignment Assign product Assign quotient Assign remainder (modulus) Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	14	Right to left
Comma	separator as in int a, b, c ;	15	Left to right

3.15.13.2 Type conversions in expressions

Automatic type conversion

It is permitted in C to mix constants and variables of different types in an expression, but it is required to convert the types according to very strict rules of type conversion. Computer considers one operator at a time, involving two operands. The conversion rule is **"if the operands are of different types, the smaller type is automatically converted to higher(wider) type before the operation proceeds."** This is called type promotion or automatic type conversion or implicit conversion. The figure 3.6 shows the rules that are applied while evaluating expression in pictorial form. Whenever a char or short int appears in an expression, it is coverted to an int. This is called **integral widening conversion.** The implicit conversion is applied only after completing all integral widening conversions.

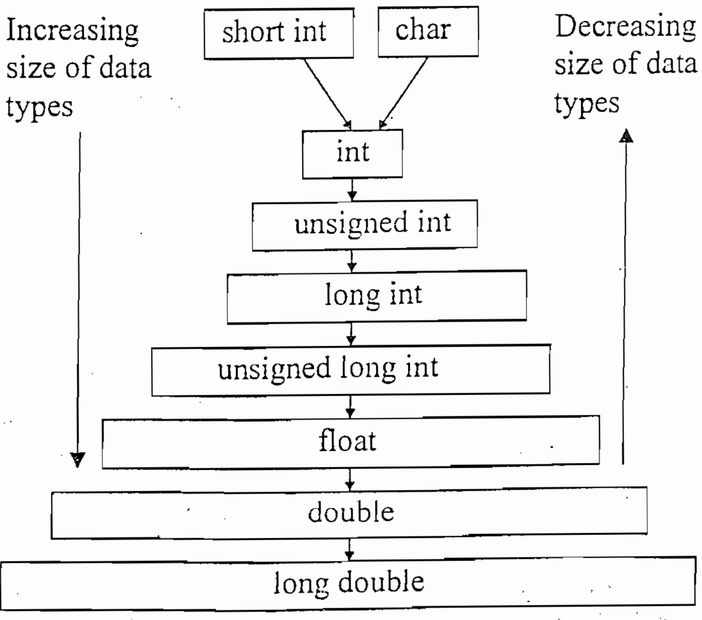


Fig.3.6 Water-fall model of type conversion

Example 3.9: Following example shows the concept of automatic type conversion.

Consider the following statements:

```
int i,y;
long int l;
float f;
double d;
```

expression y=1*f–i*d +f is illustrated in figure 3.7.

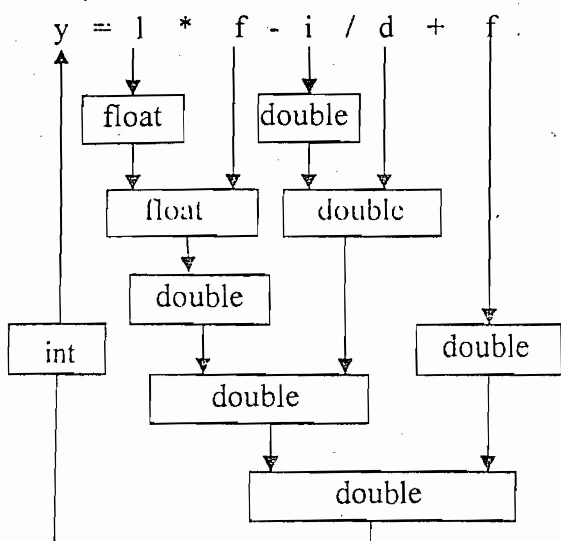


Fig.3.7 process of automatic type conversion

Question : Why is **double** converted to **int** at the final step of evaluation?
Answer : Finally, **double** is required to assign to **int** variable(y) on the left hand side of assignment operator. While assigning variables of higher types to those of lower type result in truncation. However, the following changes are introduced during the final assignments.

- **float** to **int** causes truncation of the fractional part.
- **double** to **float** causes rounding of digits.
- **long int** to **int** causes dropping of the excess higher order bits.

Type of results of mixed-mode arithmetic operations are summarized in table 3.24. How to study the table? See the highlighted column and row. Which shows if the RHO is long int and LHO is float then the result is float i.e., long int is converted to float because float is higher type than the long int.

Table 3.24 Result of mixed mode operations

RHO→ **LHO (↓)**	char	short	int	**long int**	float	double	long double
Char	int	int	int	**long int**	float	double	long double
Short	int	int	int	**long int**	float	double	long double
Int	int	int	int	**long int**	float	double	long double
long int	long int	long	long	**long int**	float	double	long double
Float	**float**	**float**	**float**	**float**	**float**	**double**	**long double**
Double	double	double	double	**double**	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO - Right Hand Operator, **LHO** - Left Hand Operator

Type casting

Sometime it is required to convert the type of an expression forcefully. This is called type casting or explicit type conversion. Whose general form is **(type-name) expression.** Where type name is one of the standard C data type. The expression may be a constant, variable or any complex expression. For example, if we want to find the ratio of TU engineering colleges to other university engineering colleges.

Suppose, TU_Engg_Cols=6
OU_Engg_Cols= 20
ratio= TU_Engg_Cols / OU_Engg_Cols
=6/20 =0 (because of integer division)
is it our objective? Obviously no. To solve this problem, we need to use type casting. Which can be done as follows:
ratio= (float)TU_Engg_Cols / OU_Engg_Cols
= 6.0/20 = 0.3

- TU_Engg_Cols is converted to float i.e., 6 to 6.0 explicitly.
- OU_Engg_Cols is converted to float implicitly i.e.,20 to 20.0. Now expression becomes 6.0/20.0

It is not always required to convert the LHO only. RHO can also be converted. Note: The variable **ratio** should be of float type.
Some examples of casts and their actions are shown in table 3.25

Table 3.25 use of casts

Exampe	Action
Y=(float) 5 = 5.0	5 is converted to float
R=(int)45.8/(int)15.5 =45/15=3	45.8 is converted to 45 by truncation. Similarly,15.5 to15.
S=(float) total/n	Division is done in floating mode.
X=(int)(p+q*s)	The result of the expression is converted to integer.
X=(int)p +q	P is converted to int and added to q
D=sin((double)x)	Covers x to double before using it

3.15.14 Some computational Problems

Errors in mixed mode arithmetic

In mixed mode expression, there may occurs some computational problems. We know that the computer gives approximate values for real numbers and the errors due to such approximation may lead to serious problems. For example, consider the following statements:

a=3/4.5;
b=a*4.5;

Mathematically, the value of b is equal to 3. But it is not guaranteed that the value of b computed in the program will be equal 3.

Division by zero

Division by zero in any point of the program leads to the abnormal program termination. In some cases such a division may produce meaningless results. It is required to avoid division by zero.

Overflow and underflow

In general usage, an overflow occurs when the volume of a substance exceeds the capacity of its intended container. In a digital computer, the condition that occurs when a calculation produces a result that is greater than that a given storage location can store or represent. Underflow occurs when a calculation produces a result that is less than a given storage location can store or represent. In other words, the number is too small to be properly stored in its allocated memory. Underflow is analogous to overflow except that the magnitude is too small rather than too large. It is our responsibility to guarantee that operand are of the correct type and range, and the result may not produce any overflow and underflow. By ordering operations carefully and checking operands in advance, it is possible to ensure that the result will never be larger or smaller than that can be stored.

3.15.16 Some important examples

Example 3.10 : Write a program that converts the temperature in Fahrenheit to Celsius.

```
#include<stdio.h>
void main(void)
{
        float fahrenheit,celsius;
        printf("Enter temperature in Fahrenheit:");
        scanf("%f",&fahrenheit);
        celsius=(fahrenheit-32)/1.8;
        printf("%f degree Fahrenheit= %f degree celsius",fahrenheit,celsius);
}
```

Output

```
Enter temperature in Fahrenheit: -40
-40.000000 degree Fahrenheit= -40.000000 degree celsius
```

Example 3.11: Illustration of use of Integer type data.

```
#include<stdio.h>
void main(void)
{
        unsigned int b;   /* Declaration */
        long int c;       /* Declaration */
        int a=45678;      /* Declaraation and assinment */
        b=45678;          /* Assignment */
        c=1234567890;  /* Assignment */
        printf("\nb=%u",b);
        printf("\nc=%ld",c);
        printf("\na=%d",a);
}
```

Output

```
b=45678
c=1234567890
a=-19858
```

Here, same value is assigned to b and a. In output, the value of b is printed exactly but value of a is printed other than the assigned value. Why so happened? This is because of the overflow of data. Here 45678 is not in the range of integer but is in the range of unsigned integer.

Example 3.12: Illustration of use of real type data.

```
#include<stdio.h>
void main(void)
{
        float a,x;
        double b,y;
        a=1.124367890000;
        b=1.23984321;
        x=y=5;
        clrscr();
        printf("\na=%f",a);
        printf("\na=%.16lf",a);
        printf("\nb=%lf",b);
```

```
        printf("\nb=%.16lf",b);
        printf("\nx=%f",x);
        printf("\ny=%.16lf",y);
}
```

Output

```
a=1.124368
a=1.1243678331375122
b=1.239843
b=1.2398432100000001
x=5.000000
y=5.0000000000000000
```

In this example, why the value of a(1.124367890000) is displayed as1.1243678331375122 under % **.16lf** format. This is because the variable x has been declared as a float that can store values only upto six decimal places. The value 1.23984321 assigned to b declared as double has been stored correctly but the printed value is 1.239843 under %lf format. This is because if there is not format specification, the printf function will always display a float or double value to six decimal places. (see chapter 4 for more about format specification)

Example 3.13 : Write a program that inputs seconds as input and converts to minutes.

```
#include<stdio.h>
void main(void)
{       int seconds,min;
        printf("Enter number of seconds:");
        scanf("%d",&seconds);
        min=seconds/60;
        seconds=seconds%60;
        printf("\nMinutes=%d",min);
        printf("\nSeconds=%d",seconds);
}
```

Output of example 3.13

```
Enter number of seconds:125
Minutes=2
Seconds=5
```

Exercise 3

1. Briefly discuss about C character set.
2. Differentiate between identifiers and keywords with example.
3. List at least 32 key words of C.
4. Explain the different types of constants in C.
5. What is ASCII value of a character? What is the role of ASCII values in information interchange?
6. What are the ASCII values of a and A? How can the ASCII value of a be obtained from the ASCII value of A. List all ASCII characters with their ASCII code.
7. Why characters proceeded by \ are called a escape sequences? Why are they required?
8. What are the ways of giving values to variables? Explain with examples.
9. Draw a diagram that shows the structure of a C program and briefly explain each part.
10. What are header files? Give some examples.
11. What are library functions? Write any 10 examples of library functions.
12. How can you define symbolic constants? Explain with a suitable example.
13. Why are enum and typedef required?
14. What are the data types available in C. Write about their size in memory, their range and the digit precision for each data type. [062/b/4-a]
15. What data types and type qualifiers are available in C? Explain with examples. [2059/c/2-a]
16. Explain in brief about the different types of operators available in C language. [061/b/2-a, 61/p/2-a]
17. Explain operator precedence and associativity with suitable example. [2059/p/2-a]
18. Explain the precedence and associativity of arithmetic operations with examples.[060/p/2-a]
19. List all the operators with their type, precedence and associativity
20. What is the difference between automatic type conversion and type casting? Explain them with examples.
21. List some computational problems while evaluating a arithmetic expression.

Chapter 4
Input and Output

4.1 Introduction

Each C program is a system. Which takes some input, process it and gives the outputs. In this chapter, we will discuss about how a C program can take inputs form standard input devices (keyboard) and displays the output on the Video Display Unit (standard output device). As a programming language C doesn't provide any input/output statements as part of its syntax. Instead a set of library functions is provided. There are a number of I/O functions in **stdio.h** header files. Some of which are shown in table 3.10. The view of discussion is shown in figure 4.1. Each program that usages a standard input/output function must contain the statement **#include<stdio.h>** at the beginning of the program. But it is not necessary for some functions like **prinf, scanf, clrscr and getch** ect. which have been defined as a part of the C language. The file name **stdio.h** is an abbrevation for standard input-output header file. The instruction **#include<stdio.h>** commands the compiler to search and insert its contents at this point in the program. Now the length of the program increases because contents of the header file becomes the part of the program. In this chapter, we will discuss different types of standard I/O functions.

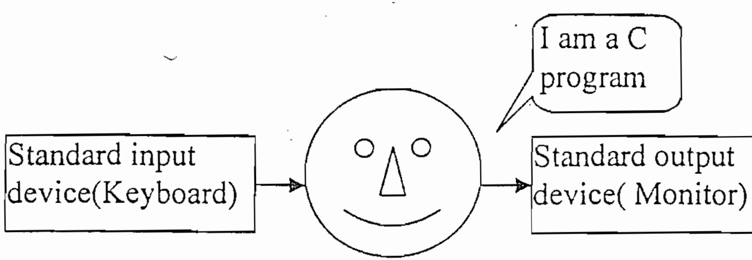


Fig. 4.1 Illustration of position of C program, input and output units

4.2 Single character input/output

getchar function reads a character from the standard input device, while **putchar** writes a character to the standard output device.

Example 4.1: A program shows a single character input/output.

```
#include<stdio.h>
void main()
{
        char ch;
        printf("Enter a character:");
        ch=getchar();
        printf("\nThe entered character is:");
        putchar(ch);
}
```

Output

```
Enter a character: a
The entered character is: a
```

4.3 String input/output

Functions **gets** and **puts** are used for inputting and outputting string.

a. gets- This function accepts string variable(the name of a string) as a parameter, and fills the string with characters that are read from the keyboard, till a new line character is encountered(i.e., till we press the enter key.) At the end the string gets appends a null character and returns the string. The new line character is not added to the string.

b. puts- puts displays the string stored in the string variable.

Example 4.2: This example illustrates the use of gets and puts.

```
#include<stdio.h>
#include<conio.h>
void main(void)
{
        clrscr();
        char str[40];
        printf("Enter your name:");
        gets(str);
        puts(str);
        getch();
}
```

Output

```
Enter your name: Sita Sapkota
Sita Sapkota
```

In this example **stdio.h** is included for function **printf**. Similarly, **conio.h** (console input/output header file) for functions **clrscr** and **getch**. Function **clrscr** clears the screen . Function **getch** is a character input function and it waits until a character is pressed on the keyboard. But it is not required to include **stdio.h** and **conoi.h** in the above example. **str[40]** means the size of string is of 40 characters. Function **gets** read a sequence of characters from the standard input device and store them in string **str**. In the sequence, the number of characters should not be greater then 39 because each string must be terminated by a null character. Function **puts** displays the string stored in the string **str**. More about string will be discussed in chapter 7.4(string).

It is necessary for programmers to give attractive and clear features with good alignment and spacing of output produced by the program. It can be accomplished by formatting output. For this purpose, there is a well-known library function named **printf** in stdio.h header file. Its general form looks like:

printf("control string",arg1,arg2,arg3,......argn);

Where control string refers to a string that contains formatting information. The formatting information consists of following items:

- Format specifications that define the output format of each data item for display.
- Characters required to make programs more interactive. These will be printed on the screen as they are in the control string
- Escape sequence characters e.g., \n, \t etc

The **arg1, agr2...argn** are the arguments that represents the individual output data items. These data items may contain constants, variable or more complex expression whose values are formatted and printed according to the specification of the control string. The arguments should match in number, order and types with the format specifications. A general form of format specification is as follows:

%-w.p conversion character

Each part is described in table 4.1.

Table 4.1 description of each part of format specification

Part	Description
%	First character of format specification. (compulsory)
- (flag)	Minus sign for left justification. It is actually the place of flags. (optional)
w (width specifier)	w is an integer number that specifies the total number of columns for the output value. (optional)
.	Period separating width from precision. (optional)
p (precision specifier)	P is also an integer number that specifies the number of digits to the right of the decimal point of a real number or number of characters to be printed from a string. (optional)
conversion character	There may be a character according to data type to be printed. For example c,d, s, f for character, integer, string and float simultaneously.(compulsory)

4.4.1 Flags

In addition to the field width, the precision and conversion character each format specification can contain a flag which affects the appearance of the output. The flag must be placed immediately after the percent sign (%). More commonly used flags are listed in table 4.2.

Table 4.2 Description of flags

Flag	Meaning
-	Data item is left justified within the field
+,-	A sign will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
0	Causes leading zeros to appear instead of leading blanks. Applies only to data item that are right justified within a field whose minimum size is larger than the data item.
' '	A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.

Commonly used format specifications for data output are listed in table 4.3

Table 4.3 Commonly used format specifications for data output *(for printf function)*

format specifier	Type of argument	Output
%c	character	Single character
%s	string	Prints characters until a null character is encountered
%d, %l	integer	Signed decimal integer
%u	unsigned integer	Unsigned decimal integer
%f	floating point	Floating point value without an exponent
%e	floating point	Floating point value with an exponent
%g	floating point	Floating-point value either in e or f forms based on given value and precision.
%E	floating point	Same as e; with E for exponent
%G	floating point	Same as g; with E for exponent if e format is used.
%ld, %li	long integer	Decimal signed long integer
%lu	unsigned long integer	Decimal unsigned long integer
%hd, %hi	short integer	Decimal signed short integer
%hu	unsigned short integer	Decimal unsigned short integer
%le, %lf, %lg	double	Double precision floating point
%Le, %Lf. %Lg	long double	Floating point with precision more than in double.
%o	integer	Octal integer, without a leading 0
%x	integer	Hexadecimal integer (with a,b,c,d,e,f)

4.4.2 Printing integer numbers

The general form of format specification for printing integer number is **%wd**. This means the integer number will be printed in w columns (field width) with right justification. Example 4.3 shows different types of format specifications to print a integer number on the screen.

Example 4.3: This example illustrates different format specifications for printing integer numbers.

```
void main()
{
        int a=12345;
        printf("\ncase 1 %d",a);
        printf("\ncase 2 %i",a);
        printf("\ncase 3 %15d",a);
        printf("\ncase 4 %-15d",a);
        printf("\ncase 5 %015d",a);
        printf("\ncase 6 %-+15d",a);
        printf("\ncase 7 %3d",a);
}
```

Output and discussion (The grid in the output can not be seen. It is used to clarify the objective of this chapter.)

Column→	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Comment
Case 1	1	2	3	4	5											Without any field width specification in %d format
Case 2	1	2	3	4	5											Without any field width specification in %i format
Case 3											1	2	3	4	5	With field width, default justification to right
Case 4	1	2	3	4	5											Using flag – to make left alignment
Case 5	0	0	0	0	0	0	0	0	0	0	1	2	3	4	5	Using flag 0 to fill the blanks before the number
Case 6	+	1	2	3	4	5		-								Number preceded by a + sign in left alignment
Case 7	1	2	3	4	5											What happen when width is less than the number of digits to be printed ?

Output and discussion

Column→	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Comment
Case1	1	2	3	.	9	8	7	6	0	2						Occupies minimum field to print in f format
Case2	1	.	2	3	9	8	7	6	e	+	0	2				Occupies minimum field to print in e format
Case3	1	2	3	.	9	8	8									Occupies minimum field to print in g format, vary n and observe various g formats.
Case4								1	2	3	.	9	8	7	6	Occupies 15 columns, prints n with 4 decimal places.
Case5	-	1	2	3	.	9	8	8								Occupies 15 columns, prints –n with left justification.
Case6	0	0	0	0	0	1	.	2	3	9	9	e	+	0	2	Prints in exponential form with 0 padding in leading blanks.
Case7	1	2	3	.	9	8	7	6	0	2	2	3				What happen when field width is not specified?
Case8	1	2	3	.	9	9										What happen when w < required field width.

4.4.3 Printing real numbers

The general form of format specification for printing real number is **%w.pf** . This means the number will be printed in w columns (field width) on the screen with p digits after the decimal point (precision) with right justification. The value, when displayed, is rounded to p decimal places. The default precision is 6 decimal places. The number will be displayed in the form [-] pppp.qqq. The general form to print real number in exponential form is **%w.p e**. The display takes the form [-] p.qqqe[±]xx. Where number of q's depends on p. The default precision is 6. The field width w should satisfy the condition $w \geq p+7$. The value rounded off and printed in the field of w column with right justification. Example 4.4 illustrates the different format specifications for printing real numbers.

Example 4.4: This example illustrates different format specifications for printing real numbers.

```
void main()
{
    float n=123.9876;
    printf("\ncase 1 %f",n);
    printf("\ncase 2 %e",n);
    printf("\ncase 3 %g",n);
    printf("\ncase 4 %15.4f",n);
    printf("\ncase 5 %-15.3f",-n);
    printf("\ncase 6 %015.4e",n);
    printf("\ncase 7 %.8f",n);
    printf("\ncase 8 %2.2f",n);
}
```

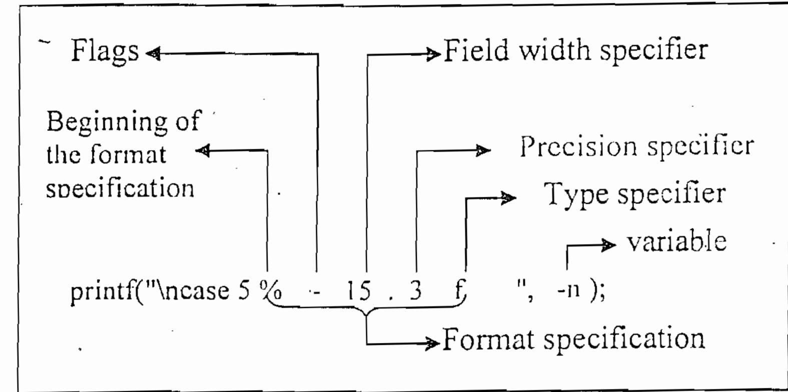


Fig. 4.2 Illustration of parts of format specification

4.4.4 Printing a single character

The general form of format specification to print a single character in a desired position on the screen looks like: **%wc**. This means a character will be printed on the field width of w columns with right justification. Example 4.5 shows some format specification for printing a single character.

Example 4.5: This example illustrates different format specifications for printing characters.

```
void main()
{
    char ch='a';
    printf("\nCase 1 =%c",ch);
```

```
    printf("\nCase 2 =%10c",ch);
    printf("\nCase 3 =%-10c",ch);
}
```

Output
Case 1 =a
Case 2 =   a
Case 3 =a

4.4.5 Printing strings

Printing strings is similar to real numbers. The general form of printing string on the screen in a field width of w columns is %w.ps. It will print first p characters of the string in the specified field width with right justification. Example 4.6 illustrates some format specifications of printing string.

Example 4.6: This example illustrates different format specifications for printing strings.

```
void main()
{
    char str[20]="I love Baglung.";
    printf("\nCase 1 %s",str);
    printf("\nCase 2 %18s",str);
    printf("\nCase 3 %-18s",str);
    printf("\nCase 4 %18.8s",str);
    printf("\nCase 5 %-18.9s",str);
    printf("\nCase 6 %5s",str);
    printf("\nCase 7 %.10s",str);
}
```

Output and discussion

Column→	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Comment
Case1	I		l	o	v	c		B	a	g	l	u	n	g	.				Occupies minimum field width to print all characters of the string.
Case2				I		l	o	v	c		B	a	g	l	u	n	g	.	Occupies 18 fields and prints string with right justification
Case3	I		l	o	v	e		B	a	g	l	u	n	g	.				Occupies 18 fields and prints string with right justification
Case4											I		l	o	v	e		B	Prints only first 8 characters with right justification.
Case5	I		l	o	v	e		B	a										Prints only first 9 characters with left justification.
Case6	I		l	o	v	e		B	a	g	l	u	n	g	.				What happen when w < minimum required field width
Case7	I		l	o	v	e		B	a	g									What happen when w is absent?

4.4.6 Printing mixed data

More than one data type can be printed using function **printf**. Example **4.7** clarifies the concept more clearly.

Example 4.7: This example illustrates the concept of printing mixed data.

```
void main()
{
        int n=12345;
        float m=123.9876;
        char ch='a';
        char str[20]="I love Bglung.";
        printf("n=%7dm=%12.5fch=%-2cstr=%16s",n,m,ch,str);
}
```

Output

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
n	=			1	2	3	4	5	m	=	.			1	2	3	.	9	8	7	6	0	c	h	=	a	
29	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**
s	t	r	=			I		l	o	v	e		B	g	l	u	n	g	.								

4.5 Formatted input

The well-known function for formatted input is **scanf**. Its general form is as follows:
scanf("control string" ,arg1, arg2, ... argn)
Where control string refers to a string containing certain formatting information of input data item. It may include:

- Format specifications, consisting of a percentage sign (%), field width (optional) and a conversion character. Format specification is the compulsory part of the control string.
- Blank, tab(\t) and new line(\n) characters. These are optional in control string. If these are used in the control string to separate the different format specifications, they must be entered consecutively while inputting the data but they will not be stored in memory.

Arguments **arg1, arg2...argn** are the arguments that represents the individual input data items. The data items may be variables. Each variable name must be preceded by an ampersand(&, address operator)except string(array of characters, see chapter 7.4(string). Actually, the arguments represent address of the data items to store them within the computer's memory. The number of format specification must equal the number of arguments and type of argument must match the corresponding format specification. **To read values of variable using scanf function, there must be at least one white space character between values of two variable. It is required to separate values of variables by any arbitrary white space while typing the values of variables. Reading value of a variable terminates at the encounter of a white space. This is the important feature of scanf function.** Basic format specification for commonly used data type for **scanf** function are shown in table 4.4

Table 4.4 Commonly used format specifications for data input (for scanf function)

Basic format	Data type	Type of argument

specification		
%c	Character	Address of a character variable
%s	String	**Name** of the string variable(why ? see chapter 7 string)
%d,	Integer	Address of an integer variable
%i	Decimal, octal or hexadecimal integer	Address of an integer variable
%u	Unsigned integer	Address of an unsigned integer variable
%f	Floating point	Address of a floating point variable
%e	Floating point	Address of a floating point variable
%g	Floating point	Address of a floating point variable
%E	Floating point	Address of a floating point variable
%G	Floating point	Address of a floating point variable
%ld, %li	Long integer	Address of a signed long integer variable
%lu	Unsigned long integer	Address of a unsigned long integer variable
%hd, %hi	Short integer	Address of a short integer variable
%hu	Unsigned short integer	Address of a unsigned short integer variable
%le, %lf, %lg	Double	Address of a double precision floating point variable
%Le, %Lf, %Lg	Long double	Address of a long double variable.
%o	Octal integer	Address of an integer
%x	Hexadecimal integer	Address of an integer
[...]	String	Name of string. [It is used to read string including white spaces.]

4.5.1 Inputting (scanning) integer numbers

The format specification for reading an integer looks like **%wd**. It means, **scanf** function reads an integer number of field width w. See example 4.8 and 4.9 for more clarification.

Example 4.8: This example illustrates different format specifications for reading integer numbers.

```
void main(void)
{
        int a,b;
        printf("Enter an integer number:");
        scanf("%d",&a);
        printf("The read and stored value of a is =%d",a);
        printf("Enter another integer number:");
        scanf("%3d",&b);
        printf("The read and stored value of b is =%d",b);
}
```

Output and discussion

Enter a integer number:12345 The read and stored value of a is =12345	No field width is specified in format specification. In such case, the given value 12345 is within the range of the integer. Therefore, it is read and assigned to memory location specified by **&a** .
Enter a integer number:12345 The read and stored value of b is =123	Field width of 3 columns. Therefore, value in the first three field is read and stored in memory location specified by **&b**.

Example 4.9: This example also illustrates different format specifications for scanning integer numbers.

```
void main(void)
{
        int a,b;
        printf("Enter two integer numbers:");
        scanf("%3d%5d",&a,&b);
        printf("\nThe read and stored value of a is =%d", a);
        printf("\nThe read and stored value of b is =%d", b);
}
```

Output and discussion

Run1	Enter two integer numbers:12 4567 The read and stored value of a is =12 The read and stored value of b a is =4567	Specified field width for a is 3 columns and b is 5. So, First number have only two digits so that it is read and stored in memory location specified by **&a**. Similarly, 4567 to **&b**
Run2	Enter two integer numbers:12345 567 The read and stored value of a is =123 The read and stored value of b is =45	Here, only first three digits(123) are read and stored in memory location specified by &a and remaining two digits (45) are read and assigned to b. Second number 567 is ignored because value of both a and b have already assigned.
Run3	Enter two integer numbers:123 1234567 The read and stored value of a is =123 The read and stored value of a is =12345	Here, first number has 3 digits, it is read and assigned to a but second number has 7 digits. Only first five digits are read and stored in memory location specified by **&b**.

4.5.2 Inputting real numbers

Reading real number is easier than the integer number because it is not required to specify the field width. The different format specifications are listed in table 4.4. Using these specifications, any desired data can be read by using **scanf** function. For more understanding see example 4.10

Example 4.10: This example also illustrates different format specifications for inputting different type of real numbers.

```
void main(void)
```

```
{
        float a,b;
        double x,y;
        long double g,h;
        printf("Enter values of a and b:");
        scanf("%f%e",&a,&b);
        printf("Value of a=%f and value of b=%e",a,b);
        printf("\nEnter values of x and y:");
        scanf("%lf%lf",&x,&y);
        printf("\nValue of x=%lf and value of y=%lf",x,y);
        printf("Enter value of g and h:");
        scanf("%Lf%Lf",&g,&h);
        printf("value of g=%Lf and value of h=%Lf",g,h);
}
```

Output

Enter values of a and b: 123.456 12.34e-13
Value of a=123.456001 and value of b=1.234000e-12
Enter values of x and y: 1234567890.12345678 1234.567e+123
Value of x=1234567890.123457 and value of y=1.2345670000000000300000000000000000000000e+126
Enter value of g and h: 12345678901234.54321111 1.1e+4932
Value of g=12345678901234.543211 and value of h=1.1000000000000000000000000000000000000000e+4932

4.5.3 Inputting a character using scanf

We have already used a function **getchar** to read a character. It can be done using a **scanf** function also. See example 4.11.

Example 4.11: This example illustrates reading of characters using scanf function.

```
void main(void)
{
        char ch;
        printf("\nEnter a character:");
        scanf("%c",&ch);
        printf("\nvalue of ch=%c",ch);
}
```

Output

Enter a character:V
value of ch=V

4.5.4 Inputting strings

As we know a string is a sequence of characters. We have already used functions gets to read a string. There are three ways of reading strings using **scanf** function.
The first way is to use **%ws** format specification. Example 4.12 illustrates this concept.

Example 4.12: This example also illustrates different format specifications for scanning strings.

```
void main(void)
{
    char str[50];
    printf("Enter a string:");
    scanf("%10s",str);
    printf("Read string is=%s",str);
    printf("Enter a string:");
    scanf("%10s",str);
    printf("Read string is=%s",str);
   printf("Enter a string:");
    scanf("%s",str);
    printf("Read string is=%s",str);
}
```

Output and discussion

Enter a string: VVVVVVVVVVVVVVV Read string is=VVVVVVVVVV	In this case, only first 10 chracters are read and stored in the array (see chapter 7) of character.
Enter a string: Nepal is a Himalayan country. Read string is=Nepal	Only first word is read in spite of other characters in the field width. This is because scanf terminates reading at the encounter of any arbitrary white space.
Enter a string: iiiiiiiiiiiiiiiiiii Read string is=iiiiiiiiiiiiiiiiiii	Reads all the characters until a white space is encountered in %s format. But the numbers of characters must be less than 50 .

The great limitation of **scanf** function to read string is that it can not read the string having multiple words because %s terminates reading at the encounter of any arbitrary white space. There are different techniques to solve this problem. Which are discussed in the following part.

Another way to read string is to specify the field width in %c format. Whose format specification is **%wc**. It means reading characters in the field width **w**. When this format is used for reading a string, the system will wait until the wth character is keyed in. Example 4. 13 illustrate this concept.

```
void main(void)
{
        char str[50];
        printf("Enter a string:");
        scanf("%10c",str);
        printf("Read string is: %s", str);
}
```

Output

Run1	Enter string: mmmmmmmmmmmmm Read string: mmmmmmmmmm	Reading only first 10 characters of the given string
Run2	Enter a string: Nepal is a Himalayan country. Read string is: Nepal is a	Reading first 10 characters with white spaces characters also. It slightly sol the problems of reading white spaces in string but it is required to fix number of characters at compile time.

The general format specifications of another way of reading string are **%[characters]** and **%[^characters]**. The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. The reading of the string will be terminated at the encounter of any other character in the input string. The specification **%[^characters]** means exactly reverse of **%[characters]**. This means the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters. The set of characters inside the brackets is called **search set.** String with white spaces can be read using these specifications. Example 4.14 and 4.15 illustrates this concept more clearly.

Example 4.14: This example shows the concept of defining search set to read strings.

```
void main()
{
    char str[70];
    printf("How old are you:");
    scanf("%[a-z0-9 ]",str);
    printf("Read string is : %s",str);
}
```

Output

How old are you: i am 21 years old. And you? Read string is : i am 20 years old	Reading is terminated at the encounter of full stop(.) because it is not in the search set.

Example 4.15: This example also shows the concept of defining search set to read strings.

```
void main()
{
    char str[70];
    printf("Enter a string:");
    scanf("%[^M]",str);
    printf("Read string is: %s",str);
}
```

Output and discussion

Enter a string: v./;'id&* |"k~!H,#$%iMo%n Read string is: v./;'id&* |"k~!H,#$%i	Reads all the characters until M is encountered. Similarly, we can use other character according to our requirement.

The search set for different purposes can be defined as follows:

- To read all decimal digits- %[0123456789] or %[0-9]
- To read all upper case characters -%[A-Z]
- To read all lower case characters - %[a-z]
- To read all decimal digits and alphabetic characters - %[0-9a-zA-Z]
- To read digits from 1 to 5 and lower case characters from p to z – %[1-5p-z]

Note: The character prior to the minus sign (-) must be lexically less than the one after it, i.e., inside search set A-Z is valid but Z-A is invalid.

Question : Why is gets function required?
A string can be read character-by-character using functions **getchar**. See chapter 7.4(string)

4.5.5 Mixed input

In a single scanf call more than one of data can be read. See example 4.16.

Example 4.16: This example illustrates the concept of reading mixed data input.

```
void main()
{
        char name[20];
        int roll;
        float marks;
        printf("Enter name, roll number and marks in B.E. entrance:");
        scanf("%s%d%f",name,&roll,&marks);
        printf("Name=%s\nRoll=%d\nMarks=%f",name,roll,marks);
}
```

Output

Enter name, roll number and marks in B.E. entrance: Mukesh 232 90.6

PIETERSEN

Name=Mukesh
Roll=232
Marks=90.599998

4.6 Points to be cared while using scanf functions are:

- Except control string each of the other arguments must be address of the a variable (pointer)
- Format specifications must match the arguments in order.
- Input data item must be separated by any white space and must match the variables receiving the input in the same order.
- When **scanf** encounters an invalid value of the data item being read, the reading will be terminated.
- While searching for a value, **scanf** ignores all the white spaces like tabs, new line.
- Any unread data items in a line will be considered as a part of the data input line to the next **scanf** call.
- If field width is used, it should be large enough to the size of input data.

4.7 Interactive (conversional) programming

Interaction means dialog between computer and users or programmers. To develop more user-friendly programs it is required to create dialog between them. In interaction, computer asks the questions and the users provides the answers, or vice versa. In C, such dialogues can be created by alternate use of the **printf** and **scanf** functions. It is not compulsory to create dialog. Most of the examples in this book have created dialogs.

4.8 Functions getche, getchar, getch

getchar - This function stops the program execution and waits until it receives a character entered on the keyboard followed by the Enter key. Until the user hits the Enter key the program will not continue to execute. (see chapter 10)

getch -This is a function gets a character from keyboard but does not echo to the screen. It allows a character to be entered without having to press the Enter key afterwards

getche - This is a function that gets a character from the keyboard and echoes to screen. It allows a character to be entered without having to press the Enter key afterwards.

Exercise 4

1. What are the standard input and output devices?
2. Why are gets and puts functions used?
3. Why is formatted output required?
4. Write a general format specification of printing different data types? Briefly discuss about each part.
5. What is the role of flags in formatted output?
6. Explain about the input/output functos available in C with the syntax and examples. [2063/b/3]
7. List the format specifications for commonly used data types for printf function?
8. Explain the way of formatted input of integer number with a suitable example.
9. What is the main limitation of scanf function to read string?
10. Can a string with white spaces be read using %wc format? Justify your answer with an example?
11. What do you mean by search set? Explain format specification %[characters] and %[^characters]? What is the main advantage of using these specifications in reading string? Explain it with a suitable example.
12. What type of programming is called interactive programming?
13. Write a program that reads name, roll, section and marks percentage of 5 students and displays in tabular form as given below:

Name	Roll Number	Section	Marks %

Chapter 5
Structured Programming Fundamentals

The basic components of a high level programming language such as C are its control structures. By this, we mean the way in which the programmer specifies the order in which instructions should be executed. Till this time, we have done some examples in sequential order. A program cannot in general achieve its function simply by executing one instruction after another. Sometimes, it is necessary to choose whether or not to perform a particular action or to perform the same action repeatedly. This section deals with the control structures available in C.
In C program, large number of functions are used that take data, process the data and return the result to the calling functions. A function is set up to perform a task. When the task is complex, many algorithms can be designed to achieve the same goal. Some algorithms may be simple but some may be complex. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures.

5.1 Sequential Structure (straight line flow)

Sequence simply means executing one instruction after another, in the order in which they occur in the source file. This is usually built in to languages as the default action, as it is with C. If an instruction is not a control statement, then the next instruction to be executed will simply be the next one in sequence.

5.2 Selective Structure (branching)

Selection means executing different sections of code depending on a condition or the value of a variable. This is what allows a program to take different courses of action depending on different circumstances.

5.3 Repetitive Structure (looping or iteration)

Repetition means executing the same section of code more than once. A section of code may either be executed a fixed number of times, or while some condition is true.

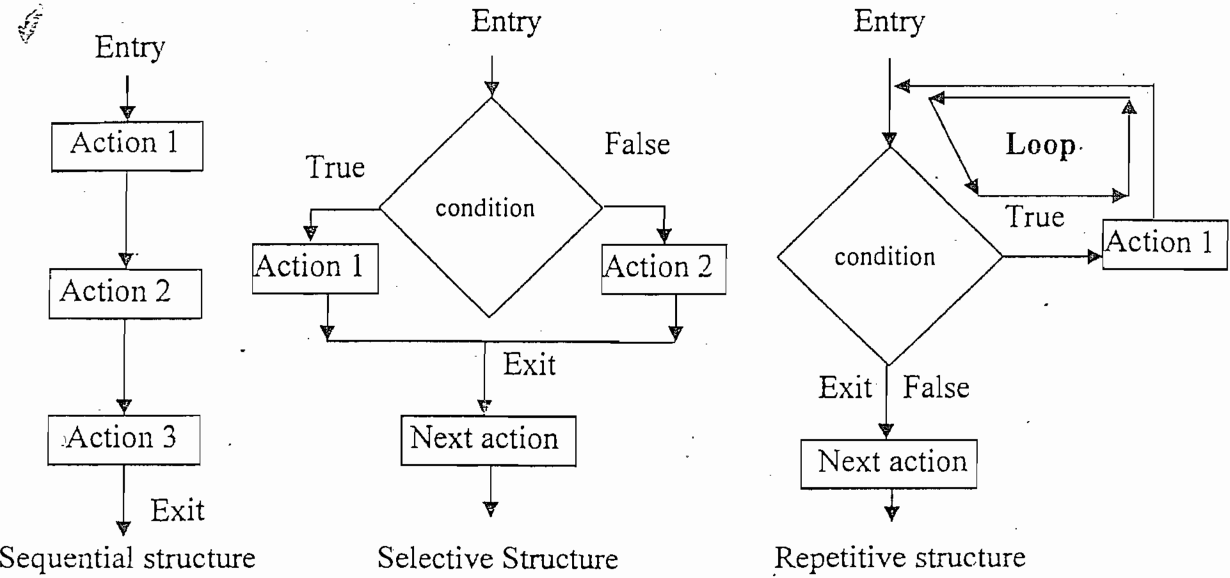


Fig. 5.1 Basic control structures

Figure 5.1 shows the execution of these control structures using single-entry and single-exit concept, which is a popular concept in modular programming. It is important to understand that entire program can be coded by using only these three logic structures. The approach of using one or more of these basic control structures in programming is known as structured programming, an important technique in software engineering. Using these basic constructs, we may represent a function structure either in detail or in summary. C supports all the three basic control structures and implements them using various control statements as shown in figure 5.2 .

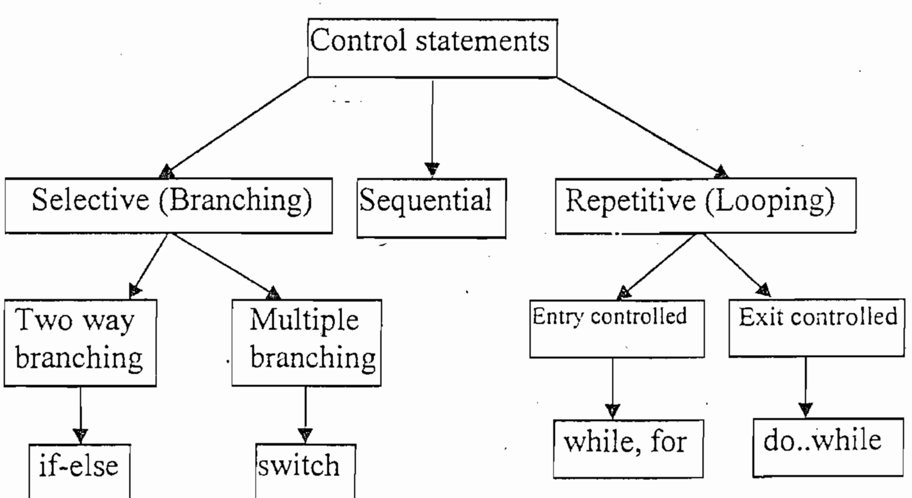


Fig. 5.2 Classification of C control statements

Let us move to study about different types of C control statements in detail based on the above-discussed basic structures.

5.1 Sequential structures

Sequential structures are the simplest structures in C. Examples 3.1, 3.2, 3.3 are in sequential structure. This structure is used when no options and no repetition of certain calculations are necessary. There is no special statement for sequential structure.

5.2 Selective structures

Selective structures are used when we have a number of situations where we may need to change the order of execution of statements based on certain condition. The selective statements must make a decision to take the right path before changing the order of execution. C provides the following statements for selective structures.

- **if statement**
- **switch statement**

5.2.1 if statement

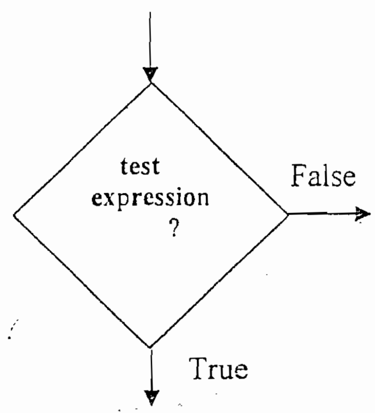
The **if** statement is a powerful decision making statement and it is used to control the flow of execution of statements. It is a two-way statement and is used in conjunction with an expression. It takes the following form.

if(test expression)

(Note: An expression which produce either true(non zero) or false(zero) is called a test expression.)

if statement allows the computer to evaluate the expression first and then depending on whether the value of the expression (condition or relation) is true or false, it transfer the control to a particular statement. At this point of the program has two paths to follow : one for the true condition and the other for the false condition as shown in figure 5.3. The if statement may be implemented in different forms depending on the complexity of conditions to be tested. Which are:

- Simple if statement
- if....else statement
- Nested if....else statement
- else....if ladder



5.2.1.1 **Fig. 5.3 two-way branching**

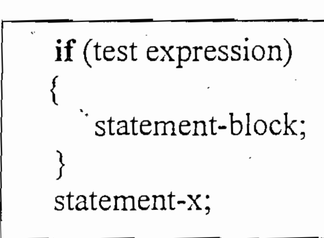
Some examples of decision-making using **if** statements are:

if(age greater than 18)
You have voting right

if(room is dark)
put on lights

if(a number is exactly divisible by 2)
The number is even

The simple **if** statement is used to conditionally execute a block of code based on whether a test condition is true or false. If the condition is true the block of code is executed, otherwise it is skipped. The general form(syntax) and flowchart for this statement is shown in figure 5.4.



```
if (test expression)
{
    statement-block;
}
statement-x;
```

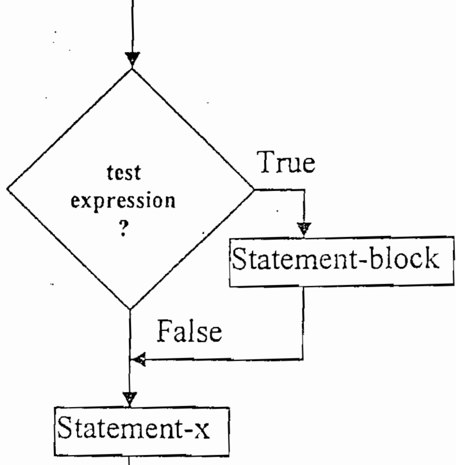


Fig. 5.4 general form and flowchart of simple if statement.

Where statement-block may be a single statement or a group of statements or nothing. If the test expression is true the statement-block will be executed otherwise the statement-block will be skipped and the execution will jump to statement-x. But, when the condition is true both the statement-block and the statement-x are executed in sequence.

Example 5.1: Write a program that prompts a user to input average marks of a student and adds 10% bonus marks if his/her average marks is greater than or equal to 65%.

```
#include<stdio.h>      /* header file for fuctions printf and scanf */
#include<conio.h>      /* header file for functions clrscr and getch */
void main(void)        /* starting of main function definition */
{
        float marks; /* variable declaration, marks is a float type variable */
        clrscr();        /* calling library function clrscr to clear the screen */
        printf("Enter marks:"); /* calling library function printf, this displays the message on the screen. */
        scanf("%f",&marks); /* calling library function scanf which takes input value of marks */
                                /* and store the value at the location allocated for variable marks in */
                            /* memory. */
        if(marks>=65)        /* if statement */
        {                                   /* starting point of body of if statement */
```

```
            marks=marks+marks* 0.1;    /* evaluation of new value of marks */
        }                              /* ending of body of if statement */
        printf("New marks=%f",marks);  /* printing new value of marks on the screen */
        getch();           /* calling library function getch which waits for any character */
}                  /* end of body of main function */
```

5.2.1.2 The if..else selection structure

The if...else statement extends the idea of the **if** statement by specifying another section of code that should be executed only if the condition is false i.e., **conditional branching.** The general form and flowchart is shown in figure 5.5. Where test expression is an expression which gives true or false value, true-block statement(s) are to be executed only if the test expression is true and false-block statement(s) is to be executed only if the condition is false. For example,

```
if (num1 > num2)
  printf("%d is greater than %d\n", num1, num2);
else
  printf("%d is not greater than %d\n", num1, num2);
```

The else clause of an **if...else** statement is associated with the closest previous **if** statement that does not have a corresponding **else** statement.

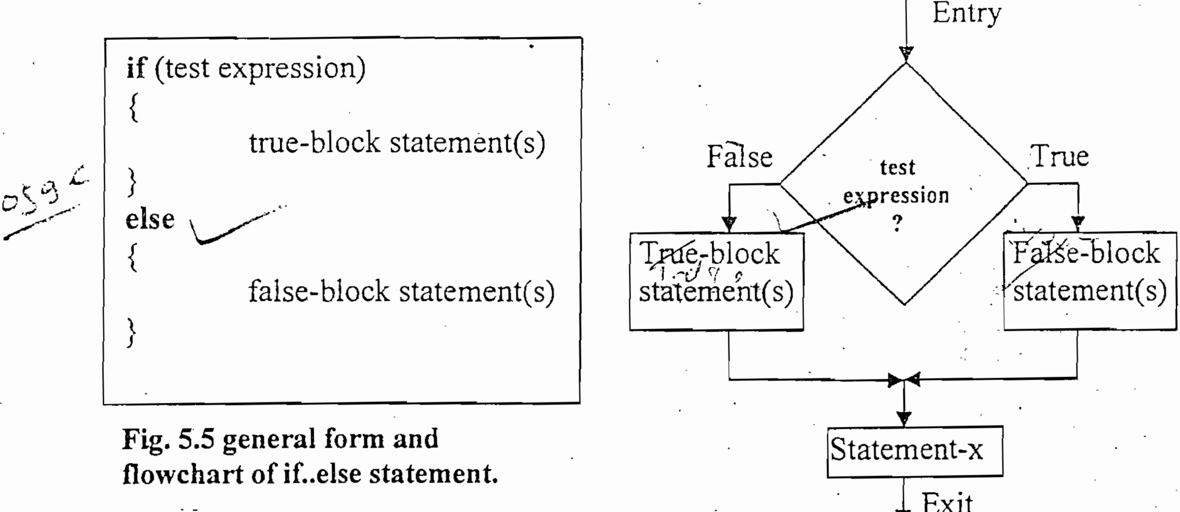


Fig. 5.5 general form and flowchart of if..else statement.

Example 5.2: Write a program to find the largest of three numbers. [058/C/1-b]

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

5.2.1.3 The nested if...else statement

When a series of decisions are involved, we may have to use more than one **if...else** statement in nested form. The ANSI standard specifies that 15 levels of nesting may be continued. In C, an else statement always refers to the nearest **if** statement in the same block and not already associated with an if. The general form and flowchart of nested **if...else** statement is shown in figure 5.6.

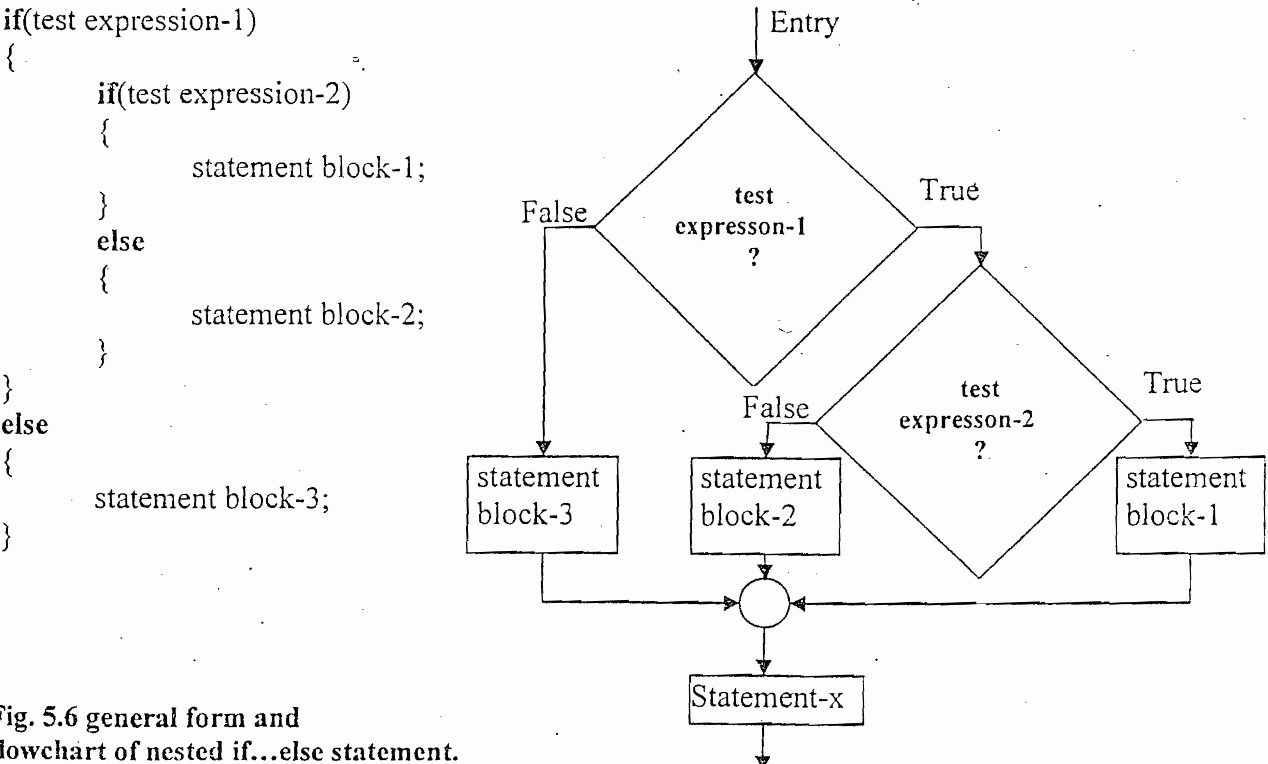


Fig. 5.6 general form and flowchart of nested if...else statement.

5.2.1.4 The else if ladder

It is another way of putting **ifs** together when multipath decisions are involved. A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**. Its general form is shown in figure **5.7**. The test expressions are evaluated from the top to downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed and control is transferred to the statement-x. If non of the conditions is true, then the final else statement will be executed. The final **else** often acts as a default condition i.e., if all other conditions' test fail, then the last else statement is executed. If there is no final **else** and all other conditions are false then no action will take place. This construct is known as **else if** ladder. The flowchart of **else...if** construct is shown in figure 5.8.

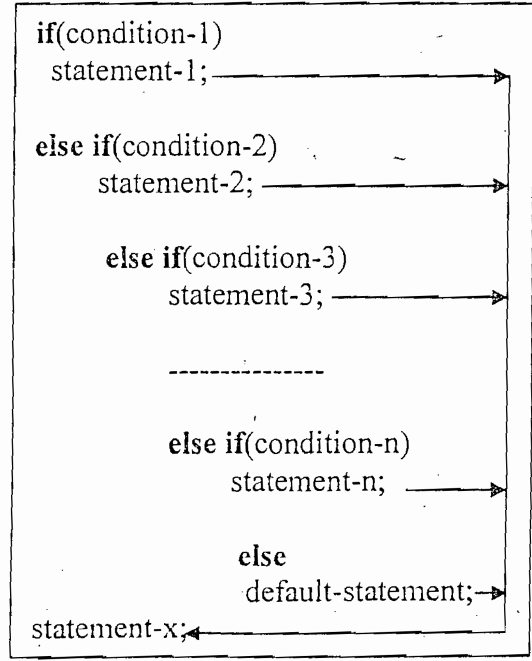


Fig. 5.7 general form of else if ladder

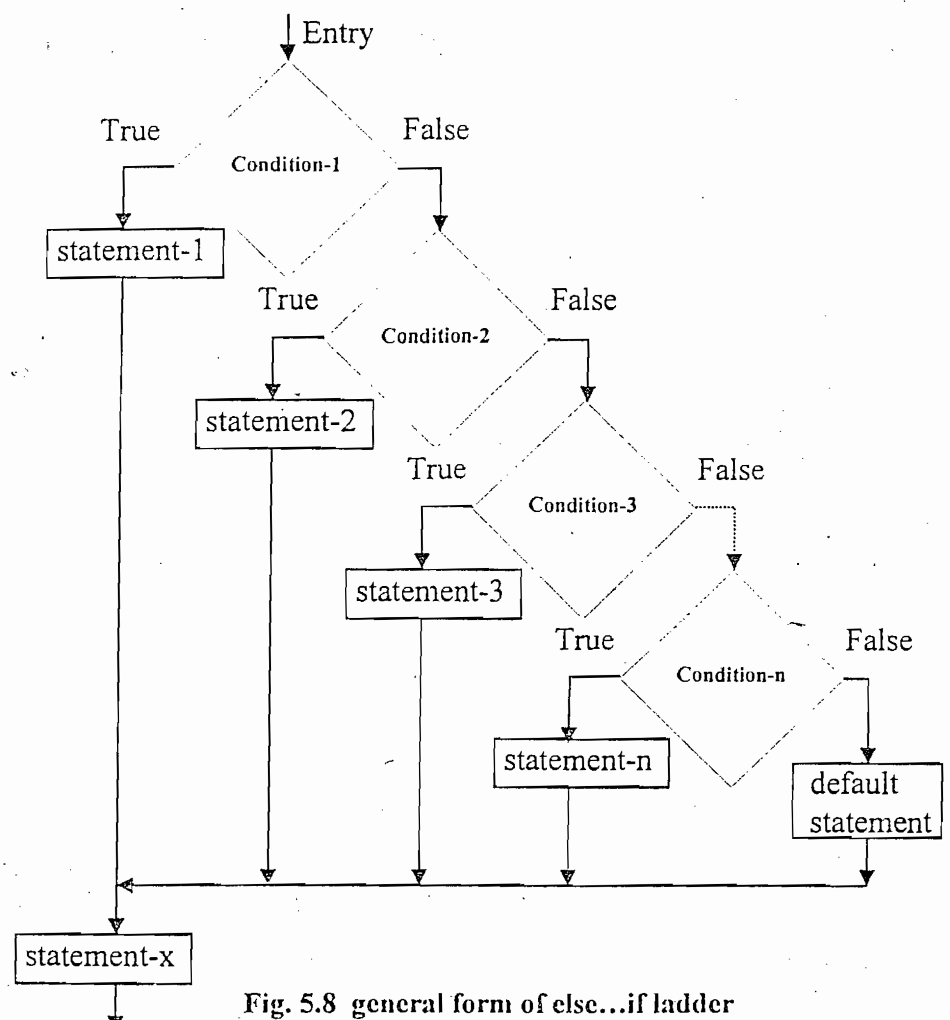


Fig. 5.8 general form of else...if ladder

Example: 5.3 This example prompts the user to enter any integer from 1 to 7 and displays the corresponding day of the week.

```
#include<stdio.h>
void main(void)
{
        int day;
        printf("Enter an integer:");
        scanf("%d",&day);
        if(day==1)
        printf("Sunday");
        else if(day==2)
                printf("Monday");
            else if(day==3)
                    printf("Tuesday");
                    else if(day==4)
                        printf("Wednesday");
                            else if(day==5)
                            printf("Thursday");
                                else if(day==6)
                                printf("Friday");
                                    else if(day==7)
                                    printf("Saturday");
                                    .else
                                        printf("Enter only 1 to 7.");
}
```

5.2.2 The switch Statement

We have seen that when one of the many alternatives is to be selected, we can design a program using **if** statements to

```
switch(expression)
{
        case constant-1:
                block-1;
                break;

        case constant-2:
                block-1;
                break;
                ...

        default:
                        default-
block;
}
statement-x;
```


Fig. 5.9 General form and flowchart of switch

control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. C has a built in multiway decision statement known as switch. It successively tests the value of an expression against a list of case values(integer or character constants). When a match is found, the statements associated with that **case** is executed. The general form and flowchart is shown in figure 5.9.

Table 5.1 Switch Statement Behavior

Condition	Action
The value of switch expression is matched to a case label.	Control is transferred to the statement following that label.
The value of switch expression is not matched to any case label and the **default** label is present.	Control is transferred to the statement following **default** label.
The value of switch expression is not matched to any case label and the **default** label is not present.	Control is transferred to the statement after the **switch** statement.

The expression is an integer or character expressions. constant-1, constant-2....are constant or constant expressions(evaluable to an integral constant) and are known as the case labels. Case labels should be unique within a switch statement. ANSI C requires at least 257 case labels be allowed in a **switch** statement. Case labels end with a semicolon (:). If character constants are used as case labels, they will be converted to integers automatically. Although case is a label statement, it cannot exist outside a switch. Block-1, block-2... are statement lists to be executed and may contain zero or more statements. There is no need to put braces around these blocks. The **break** statement at the end of each block indicates the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch. If the **break** statement is omitted, execution continues on into the next case statement until either a break or the end of the switch is reached. The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case constants. If not present, no action takes place if all the matches fail and control goes to the statement-x. It is possible to nest the switch statements. A switch may be part of a **case** block. ANSI allows 15 levels of nesting. Overall behavior of switch statement is summarized in table 5.1.

Example 5. 4: Write a program that asks an arithmetic operator and two operands and performs the corresponding operation.

```
#include<stdio.h>
void main(void)
{
        char operatorr;
        int a,b;
        float result;
        printf("Enter a arithmetic opertor +,-,* or /");
        operatorr=getche();
        printf("\nEnter value of a:");
        scanf("%d",&a);
        printf("\nEnter value of b:");
        scanf("%d",&b);
        switch(operatorr)
        {
                case '+':
                        result=a+b;
                        printf("\na+b=%f",result);
                        break;
                case '-':
                        result=a-b;
                        printf("a-b=%f",result);
                        break;
                case '*':
                        result=a*b;
                        printf("a*b=%f",result);
                        break;
                case '/':
                        result=a/(float)b;
                        printf("a/b=%f",result);
                        break;
                default:
                        printf("Enter + or - or * or /");
                        break;
        }
}
```

Output

```
Enter a arithmetic opertor +,-,* or /
Enter value of a: 10
Enter value of b: 20
a+b=30
```

Question : Solve example 2 using switch statement.

Note: More than one case labels can be used for a statement block.

For example:

```
................
ch=getche();
switch(ch)
{
        case 'a':
        case 'A':
                printf("Alphabetic character.");
                .......
        ...........
}
................
```

In the switch construct, as in the **else...if** construct, multiple possible blocks of code are being selected from and only one of them will typically be executed. The **else...if** ladder and the **switch** construct have very similar usage in a program but there are distinct differences. These differences are presented in the table 5.2.

Table 5.2 Differences between else...if and switch statement

else...if construct	switch construct
An expression is evaluated and the code is selected based on the truth-value of the expression.	An expression is evaluated and the code is selected based on the value of the expression.
Each **if** has its own logical expression to be evaluated as true or false.	Each case is referring back to the original value of the expression in the switch statement.
The variables in the expression may evaluate to a value of any type, either an int or a char.	The expression must evaluate to an int.
It does not require **break** statement because only one of the blocks of code is executed	It needs involvement of the break statement to avoid execution of the block just below the current executing block.
It takes decision on the basis of non zero(true) or zero(false) basis.	It takes decision on the basis of equality.

5.2.3 The conditional operator (? :)

This operator is used for making two way decision. It takes three operands. The general form is **conditional expression ? expression-1 : expression-2**

The conditional expression is evaluated first. If the result is non zero(true), expression-1 is evaluated and is returned as the value of conditional expression. Otherwise, expression-2 is evaluated and its value is returned. For example, the program segments:

```
        if(a>b)
                value = 2*a-b;
        else
                value = a+5*b;
```

can be written as

```
                value=(a>b) ? (2*a-b) : (a+5*b);
```

The conditional operator may be nested for evaluating more complex assignment decisions.

Example 5.5 : Write a program that finds the value of y based on the value of x in the following function.

$$y = \begin{cases} 2x+300 & \text{for } x<50 \\ 200 & \text{for } x=50 \\ 50x-100 & \text{for } x>50 \end{cases}$$

```
#include<stdio.h>
void main(void)
{
        float x,y;
        printf("Enter value of x:");
        scanf("%f",&x);
        y=(x!=50)?((x<50)?(2*x+300):(50*x-100)):200;
        printf("y=%f",y);
}
```

Example 5.6: Write a program to check whether the entered year is leap year or not.

The Gregorian calendar, the current standard calendar in most of the world, adds a 29th day to February in all years evenly divisible by 4, except for century years (those ending in -00), which receive the extra day only if they are evenly divisible by 400. Thus 1996 was a leap year whereas 1999 was not, and 1600, 2000 and 2400 are leap years but 1700, 1800, 1900 and 2100 are not.

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

5.3 Repetitive structure (iterative or loop structure)

In repetitive structure, a sequence of statement(s) which is specified once, but which may be executed zero or several times in succession. The sequence of statement(s) may be executed for a specified number of times or until some condition is met. The structures which executes the statement(s) for a specified number of times are called count controlled loops. The structures which executes the statement(s) until some condition is met are called condition (sentinel) controlled loops. A looping process, in general, would include the following four steps.

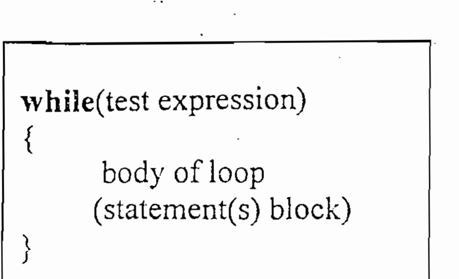
- Setting and initialization of a counter.
- Execution of statements in the loop.
- Test for a specified condition for execution of the loop.
- Updating the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met. C provides for three loop constructs for performing loop operations. They are:

- **while statement**
- **do...while statement**
- **for statement**

5.3.1 The while statement

The while statement specifies that a section of code should be executed while a certain condition holds true. The general syntax and flowchart is shown in figure 5.10.



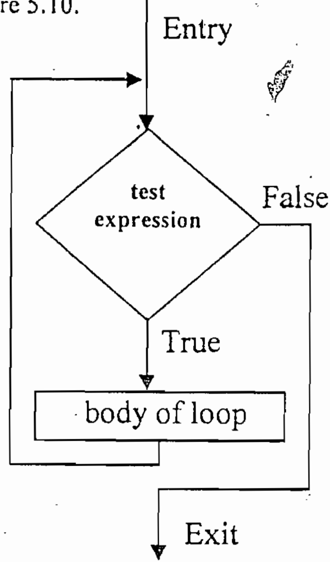


Fig. 5.10 General form and flow chart of while loop.

where test expression is an expression which gives either true(nonzero) or false(zero), and body of the loop is the statement to be repeatedly executed while the condition is true. The body of loop may have no or one or more statements. The brace are needed only if the body contains two or more statements. However, it is good practice to use braces even if the body has only one statement. At first, the test condition is evaluated and if it is true(non zero), then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop. A **while** loop can also be forcefully terminated when a **break, goto or return** within the body of the loop is executed. **Continue** statement is used to terminate the current iteration without exiting the **while** loop. **continue** passes control to the next iteration of the **while** loop.

Example 5.7: Write a program which asks the user to input positive numbers and finds the sum of all the numbers until the current sum remains less than or equal to 1000.

```
#include<stdio.h>
void main(void)
{
        int number,sum=0;
        while(sum<=1000)
        {
                printf("Enter a number:");
                scanf("%d",&number);
                sum=sum+number;
        }
        printf("Sum=%d",sum);
}
```

In the above example, initial sum is 0. When the control goes to while loop, the test expression (sum<=1000) evaluates to true because current sum(0) is less than 1000. Then, the user enters a number, which will be added to the current sum and will be assigned to the variable sum. Then, control goes back to the test expression. If the condition is true, the body of the loop will be executed again. Otherwise, control will go to the statement printf("Sum=%d",sum); and will print the sum.

Question : What will be final value of sum in the above program?
Answer : a.1000 b. >1000 c.<1000

5.3.2 do...while statement

The **do...while** statement is very similar to the **while** statement. It also specifies that a section of code should be executed while a certain condition holds true. The general syntax and flowchart is shown in figure 5.11.Where test expression is an expression which evaluates either true(nonzero) or false(zero) and body of the loop is the statement to be repeatedly executed while the condition is true. The body of loop may have no or one or more statements. The braces are needed only if the body contains two or more statements. However, it is good practice to use braces even if the body has only one statement. On reaching the **do...while** loop , the program proceeds to evaluate the body of loop first. At the end of loop, the test expression in the while statement is evaluated. If the test expression evaluates to true(non

```
do
{
        body of loop

}while(test expression);
```

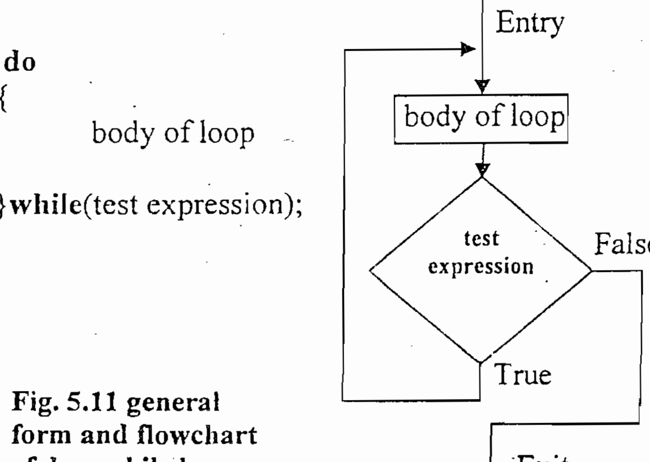


Fig. 5.11 general form and flowchart of do...while loop.

..., the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement. The **do... while** loop can also be forcefully terminated when a **break, goto or return** within the body of the loop is executed. **Continue** statement is used to terminate the current iteration without exiting the **while** loop. **continue** passes control to the next iteration of the **do...while** loop. The **do...while** loop is very similar to the **while** statement. The difference between a **do...while** loop and a **while** loop is that the while loop tests its condition at the top of its loop but the **do...while** loop tests its condition at the bottom of its loop. As noted above, if the test condition is false as the while loop is entered the block of code is skipped. Since the condition is tested at the bottom of a do loop, its block of code is always executed at least once.

Example 5.8: Example 5.7 using do...while loop. Study, understand and differentiate it with while loop.

```
#include<stdio.h>
void main(void)
{
        int number,sum=0;
        do
        {
                printf("Enter a number:");
                scanf("%d",&number);
                sum=sum+number;
        }
        while(sum<=1000);
        printf("Sum=%d",sum);
}
```

Question : What will be final value of sum in the above program?
Answer: a.1000 b. >1000 c.<1000

5.3.3 The for statement

Both the **while** and the **do...while** statements provide repetition until some condition becomes false. The for statement is used to execute a block of code for a fixed number of repetitions. The **for** statement uses a loop variable to count the number of times that the loop has been executed. The general form and flowchart of for statement is shown in figure 5.12. **Initialization** is generally an assignment statement used to set the loop control variable. **Test expression** is a relational expression that determines when the loop exits. **Update expression** defines how the loop variable changes each time the loop is repeated. Body of the loop is a set of statements to be executed repeatedly while the condition remains true. The body of the loop may have no or one or more statements. The braces are needed only if the body contains two or more statements. However, it is good practice to use braces even if the body has only one statement.

```
for(initialization expression; test expression; update expression)
{
        body of loop;
}
```

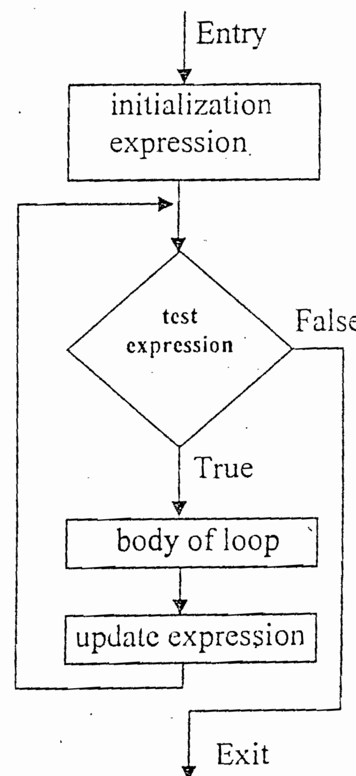


Fig 5.12 General form and flowchart of for loop

Table 5. 3 The three operational parts of for loop

Part	Why used ?	When Executed ?
Initialization expression	To initialize the loop control variable.	Before any other element of the **for** statement. Initialization expression is executed only once. Control then passes to test expression
Test expression (conditional expression)	To test for loop-termination criteria.	Before execution of each iteration of body of loop, including the first iteration. Body of loop is executed only if test-expression is evaluated to true (nonzero).
Update expression	To obtain new value of the loop control variable. Normally new value can be obtained by incrementing (decrementing) loop control variable.	At the end of each iteration of body of loop. Before execution of test expression, update expression is required to be executed to find new value of loop control variable.

For example,
for (i = 0; i < 10; i++)
The initializer (i = 0) is executed only once before the first time execution of the loop. The loop condition (i < 10) is tested each time before the loop is executed. The loop statement (i++) is executed after each execution of the loop. This means that, in this example body of the loop will be executed ten times. At the first time i will be equal to 0 and at the last time i will be equal to 9.

Example 5.9:Write a program that prints out a number of stars specified by a user.

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

Output

```
Enter the number of stars: 10
**********
```

5.3.3.1 Variation of for loop

The **for** loop in C has several capabilities that are not found in other loop constructs. Different types of variations are listed below.

I. **More than one variables can be initialized at a time using comma operator:**

For example, the statements:

```
i=1;
for(j=0;j<20;j++)
can be written as  for(i=1,j=0; j<20;j++)
```

II. **There may be more than one part in update section as in the initialization section separated by comma operator.**

For example, the statements:

```
for(i=0,j=100; i<=j;i++,j--)
{
        sum=i+j;
        printf("Sum=%d",sum);
}
```

is valid body of for loop.

III. **Test condition may have any compound relational and the testing need not be limited only to the loop control variable.**

For example, the statements:

```
sum=0;
for(i=0; i<50 && sum<500; i++)
{
```

```
    printf("%d",sum);
}
```

is valid. In the test condition of the above example, logical operator(&&) is used to relate two different relational expressions (i<50 and sum<500) and sum is not a loop control variable but used in test condition.

IV. Missing pieces of the loop definition

An interesting feature of the for loop is that pieces of the loop definition can be omitted.

For example,

a. Omission of initialization expression

To omit the initialization expression inside for statement it can be initialized before for statement. For example:

```
for(i=0;i<30;i++)
can be expressed as
i=0;
for(;i<30;i++)
```

b. Omission of initialization and test condition

Omitting test condition means test condidion is always true. In such condition, loop goes to infinite iteration. To make the it finite, we must use break statement. Study the example 5.10.

Example 5.10: Illustration of use of break statement to break an infinite loop.

```
#include<stdio.h>
void main(void)
{
        int i=0;
        for(;;i++)
        {
                if(i>20)
                break;
                printf("KEC");
        }
}
```

c. Omission of initialization expression, test condition and update expression

If all of the expressions in the loop definition are omitted, an infinite loop is created. `break` statement must be used to break out of the loop, as in:

```
for( ; ; )
{
    ch = getchar();
    if(ch = = 'A')
    break;
}
```

Omission of initialization and update expression

For example, see the statement:

```
for(i=0; i<30; i++)
Can be written as
i=0;
for( ;i<30; )
{
    .......
    i++;
}
```

Similarly, find other possible omission of expressions in for statement and find the alternative solutions.

V. for loops with no body(null statements)

The body of the for loop may also be empty. Time delay loops can be created using an empty body of for loop. For example, following loop statement delays time till t reaches to 0.

```
for(t=1000;t>=0;t--);
```

One of the important feature of the for loop is that all the three actions, namely initializing, testing and updating are placed in the for statement itself making them visible to the programmers and users in one place. The for loop can be expressed in **while** and in **do...while** loops. The **while** and **do...while** equivalent form of for loop are shown in the following table 5.4.

Table 5.4 Comparisons of the three loops

for	while	do...while

for(i=0; i<=40; i=i+1)
{
....
....
}
	i=0;
	while(i<=40)
	{
	....
	....
	i=i+1;
	}
		i=0;
		do
		{
		....
		....
		i=i+1;
		}
		while(i<=40);

Similarly, **while** loop can be rewritten as **for** and **do...while** and **do...while** can be rewritten as **for** and **while** loops but slightly more change is required. Though form of loops to use is a matter of personal choice. We can choose a loop to use properly on the basis of the following points.

- While analyzing the problem, it is required to decide whether it requires counter based control loop or sentinel based control is required.
- If the counter based control is required, use for loop.
- If the sentinel-based (condition based) control is required, use while or do...while loops.
- Note that both the counter controlled and sentinel controlled loops can be implemented by all the three looping constructs (for, while, do...while)
- According to the nature of the problem, it also requires to decide whether a testing should be done at the beginning of the loop or at the end of the loop.
- If it requires a post-test loop, we can use do...while loop.
- If it requires a pre-test loop, we can use while or for loops.

5.4 Nesting of loops

Putting one loop statement with in another loop statement is called nesting of loops.

5.4.1 Nesting of for loops

Putting one for statement within another **for** statement is called nesting of **for** loops. Two loops can be nested as follows:

```
.........
.........
for(i=0;i<10;i++)
{
        .........
        .........
        for(j=0;j<10;j++)
        {
                .........
                .........
        }
        .........
        .........
}
```

Nesting may be continued upto 15 levels in ANSI C; many compilers allow more. The program segment to print the multiplication table of rows by columns is as follows:

```
............
............
for(i=0;i<=rows;i++)
{
        for(j=0;j<=columns;j++)
        {
                product=i*j;
                printf("%d", product);
        }
        printf("\n");
}
```

In the above example, the outer loop controls the rows while the inner loop controls the columns.

5.4.2 Nesting of while loops

Above example can be written using while loop as follows:

```
............
............
i=0;
while(i<=rows)
{
        j=0;
        while(j<=columns)
        {
                product=i*j;
                printf("%d",product);
                j++;
        }

i++;
}
```

5.4.5 Nesting of do...while loops

Above example can be written using **do... while** loop as follows:

```
...........
...........
i=0;
do
{
        j=0;
        do
        {
                product=i*j;
                printf("%d",product);
                j++;
        } while(j<=columns);
i++;
} while(i<=rows);
```

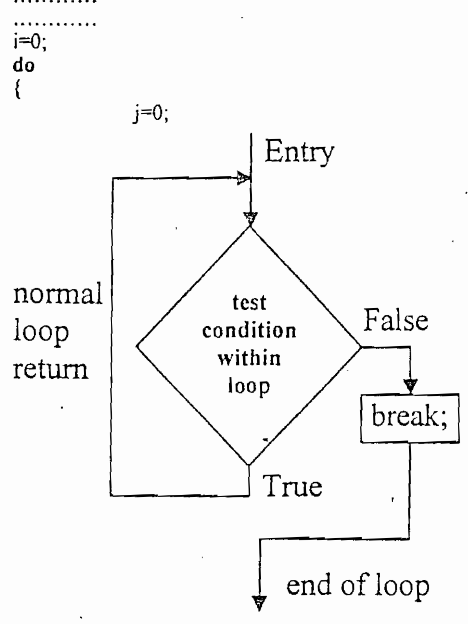


Fig. 5.14 flowchart of break statement

5. 5 Entry controlled and exit controlled loops

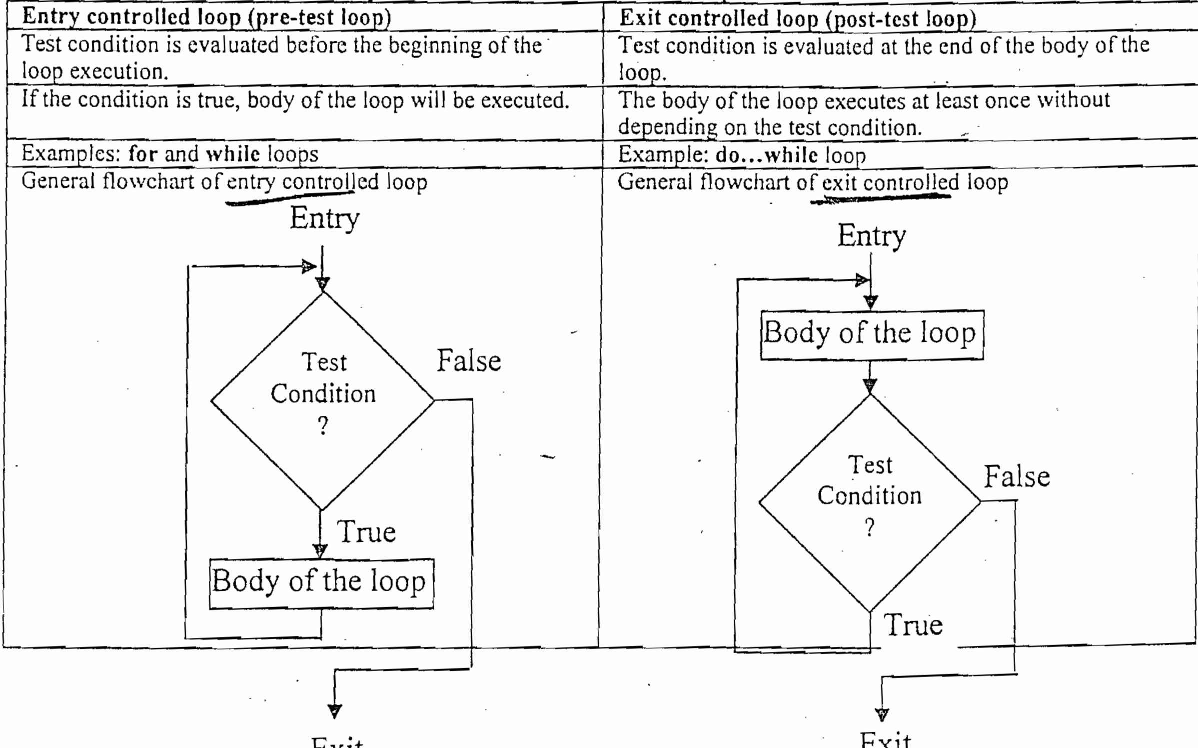
Depending on the position of control statement in the loop, a loop may be classified either as the **entry controlled** loop or as the **exit controlled** loop. These are differentiated in the following table 5.5.

5.6 Definite and indefinite iterations(loops)

There are two forms of iteration. Definite iteration is where the number of times the part of the program will be executed can be determined by the program immediately before the iterative structure is encountered. The for loop is an example of definite iteration. This includes the situation where the number of times the part of the program will be executed can be determined from the specification. An indefinite iteration is where the number of times the part of the program will be executed can not be determined until the iterative structure is encountered. While and do...while loops are example of indefinite iteration.

Table 5.5 Difference between entry controlled and exit controlled loops

Entry controlled loop (pre-test loop)	**Exit controlled loop (post-test loop)**
Test condition is evaluated before the beginning of the loop execution.	Test condition is evaluated at the end of the body of the loop.
If the condition is true, body of the loop will be executed.	The body of the loop executes at least once without depending on the test condition.
Examples: **for** and **while** loops	Example: **do...while** loop
General flowchart of entry controlled loop	General flowchart of exit controlled loop



5.7 Jumps in loops

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. C permits a jump from one statement to another within loop as well as a jump out of a loop. **break, continue, goto** and **return** statements are used for jumping purpose. First three are discussed in the following sections and the return statement will be discussed in chapter 6 (functions).

5.7.1 The break statement

Break statement is used to jump out of a loop. The **break** statement terminates the execution of the nearest enclosing loop. Control passes to the statement that follows the terminated statement, if any. **break** is used with the conditional switch statement and with the **while, do...while** and **for** loop statements. In a **switch** statement, **break** causes the program to execute the next statement after the **switch**. Without a **break** statement, every statement from the matched

case label to the end of the **switch**, including the **default**, is executed. In loops, **break** terminates execution of the nearest enclosing **while, do...while** or **for** statement. Control passes to the statement that follows the terminated statement, if any. Within nested statements, the **break** statement terminates only the **while, do...while, for** or **switch** statement that immediately encloses it. This statement is written as,

break;

The flowchart of **break** statement is shown in figure 5.14. Illustration of exiting a loop with break statement is shown in figure 5.15.

```
while(.........)
{
        .........
        .........
        if(condition)
exit    break;
form    .........
loop
}
.........
```

```
do
{
        .........
        .........
        if(condition)
        break;
exit    .........
from
}while(.........);
loop
...............
```

```
for(.........)
{
        .........
        .........
        if(condition)
exit    break;
form    .........
loop
}
.........
```

```
for(.........)
{
        for(.........)
        {
                ........
                if(condition)
exit            break;
from            .........
loop
        }
        ........
}
```

Fig.5.15 Illustration of exiting a loop with break statement

```
Exa[...]
#incl[...]
void main()
{
        {
                printf("%d\n", i);
                if (i == 4)
                break;
        }
} /* Loop exits after printing 1 through 4.*/
```

5.7.2 The continue statement

continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. The remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. This statement is written as,

continue;

The flowchart of continue statement is shown in figure 5.16.

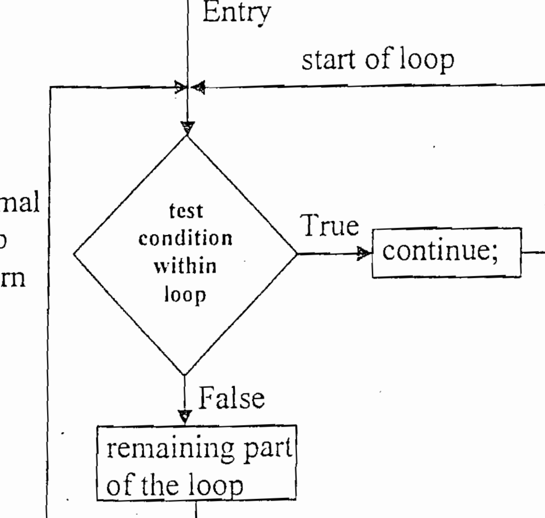


Fig.5.16 Flowchart of continue statement

The next iteration of the loop is determined as follows:

- In the **while or do...while** loop, the next iteration starts by reevaluating the test condition of the **while or do...while** statement, depending on the result, the loop either terminates or another iteration occurs.
- In the **for** loop (using the syntax **for** (initialization; **test condition; update**)), **continue** causes update expression to be executed. Then test condition is reevaluated and depending on the result, the loop either terminates or another iteration occurs.

These points are illustrated in the following figure 5.17.

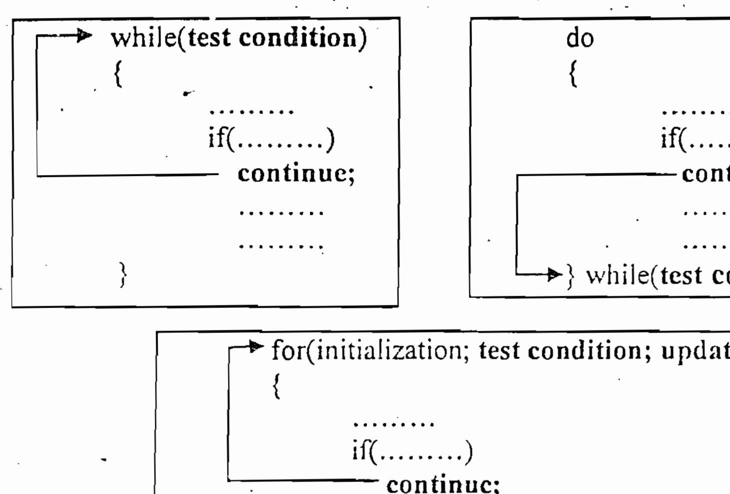


Fig 5.17 Illustration of bypassing and continuing in loops

Example 5.12: The following example shows how the continue statement can be used to bypass sections of code and begin the next iteration of a loop. This program prints the alphabetic characters and their ASCII values in the range from 65 to 122 but there are non alphabetic characters form 91 to 96 which are skipped using continue statement.

```
#include<stdio.h>
void main(void)
{
        int i;
        for(i=65;i<=122;i++)
        {
                if(i>90&&i<97)
                {
                        continue;
                }
                printf("%c=%d,",i,i);
        }
}
```

Output

A=65,B=66,C=67,D=68,E=69,F=70,G=71,H=72,I=73,J=74,K=75,L=76,M=77,N=78,O=79,P=80,Q=81,R=82,S=83,T=84,U=85,V=86,W=87,X=88,Y=89,Z=90,a=97,b=98,c=99,d=100,e=101,f=102,g=103,h=104,i=105,j=106,k=107,l=108,m=109,n=110,o=111,p=112,q=113,r=114,s=115,t=116,u=117,v=118,w=119,x=120,y=121,z=122.

5.7.3 The goto statement

The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program. The goto statement is written as

goto label;

where label is an identifier that is used to label the target statement to which the control will be transferred. Control may be transferred to any other statement with in the program.(to be more precise, control may be transferred anywhere with in the current function). The target statement must be labeled and the label must be followed by a colon. The target statement will appear as

label: statement

each label statement with in the program(more precisely, with in the current function) must have a unique label. The label can be before or after the goto statement. **goto** breaks the normal sequence of the program. If the **label:** is before the statement **goto label;** a loop will be formed and some statement will be executed repeatedly. Such a jump is known as backward jump. On the other hand, if the **label:** is placed after the **goto label;** statements some statement will be skipped and the jump is known as a forward jump. Which is shown in figure 5.18.

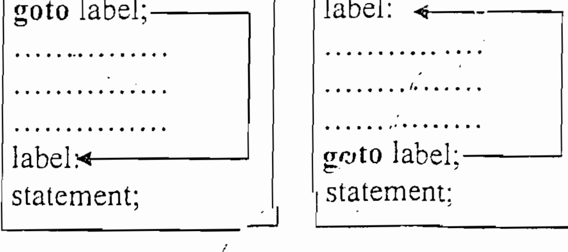


Fig 5.18 Illustration of forward and backward jumps of goto statement

Example 5.13: Following program illustrates the use of goto statement.

```
#include <stdio.h>
void main()
{
        int i, j;
        for ( i = 0; i < 10; i++ )
        {
                printf( "Outer loop executing. i = %d\n", i );
                for ( j = 0; j < 2; j++ )
```

```
            printf( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
            goto stop;
        }
    }
    printf( "Loop exited. i = %d\n", i ); /* This message will not bet printed */
    stop:
    printf( "Jumped to stop. i = %d\n", i );
}
```

In this example, a **goto** statement transfers control to the point labeled **stop** when **i** equals 3.

How to the avoid use of goto ?

The most common application of goto are:

- Branching around statements or groups of statements under certain conditions.
- Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during current pass.
- Jumping to the end of a loop under certain conditions, thus terminating the execution of a loop.

The structured features in C enable all of these operations to be carried out without using the **goto** statement. Which are as follows:

- Branching around statement can be accomplished using **if...else** statement.
- Jumping to the end of loop can be accomplished using **continue** statement.
- Jumping out of a loop can be accomplished using **break** statement.

Reasons for avoiding goto are:

- Programs developed using **goto** becomes less efficient.
- Using many of them makes program logic complicated and renders the program unreadable.

Note that using **goto** is not illegal. The effective use of **goto** statement is to transfer the control out of deeply nested loops since the **break** statement only exits from one level of the loop.

5.8 True and False and test expressions in C

Many C statements rely on the outcome of a test expression, which evaluates to either **true** or **false**. In C, true is any non-zero value and false is zero. This approach facilitates efficient coding. The programmer is not restricted to test expressions involving only the relational and logical operators. Any valid C expression may be used to produce either true or false. All that is required is an evaluation to either zero or non-zero.

Example 5.14: Illustration of using an arithmetic expression as a test expression in for statement.

```
#include<stdio.h>
void main(void)
{
        int a,b;
        scanf("%d%d",&a,&b);
        if(!(a-b))
        printf("a and b are equal.");
        else
        printf("a and b are not equal.");
}
```

In this example, if a and b are equal then (a-b) results in zero but not(!) operator is used ahead of the expression which changes the zero to nonzero. Therefore, nonzero means true which directs the execution of the if block.

5.9 Overview of control statements

Table 5.7 summary of C control statements

Category of statements	Keywords
Selection (branching)	if, switch
Repetition(iteration)	for, while, do-while
Jump	break, goto, continue, return
Label	case, default

5.10 Some important examples

Example 5.15: Write a program to read two integers n1 & n2. Display all even numbers between those two numbers. (n1<n2). [058/P/4-b]

```
#include<stdio.h>
void main(void)
{
        int n1,n2,i;
        printf("Enter two numbers:");
        scanf("%d%d",&n1,&n2);
        for(i=n1;i<=n2;i++)
        if(i%2==0)
                printf("\t%d",i);
}
```

Example 5.16: Write a program to count the number of odd and even number entered by the user. Your program must read numbers until the user enters zero. After user enters zero display the counts. [058/C/2-b, 059/C/4-b]

```
#include<stdio.h>
void main(void)
{
        int num,evencount=0,oddcount=0;
        while(1) /* always true condition, indefinite loop */
        {
                printf("Enter a number:");
                scanf("%d",&num);
                if(num==0)
                        break;  /* to break the indefinite loop */
                else if(num%2==0)
                        evencount=evencount+1;
                else
                        oddcount=oddcount+1;
        }
        printf("Number of even number=%d",evencount);
        printf("Number of odd numbers=%d",oddcount);
}
```

Example 5.17: Write a computer program that reads in an integer value for n then sums the integers from n to 2n if n is non-negative, or from 2n to n if n is negative. Display the sum. [057/3C/b]

```
#include<stdio.h>
void main(void)
{
        int n,i,sum=0;
        printf("Enter a number:");
        scanf("%d",&n);
      if(n>0)
        {
                for(i=n;i<=2*n;i++)
                sum=sum+i;
        }
        else if(n<0)
        for(i=2*n;i<=n;i++)
        {
                sum=sum+i;
        }
        else
        {
                printf("Etered number is 0");
        }
        printf("The sum is %d",sum);
}
```

Example 5.18: Write a program to find the terms in the given series till the value of term is less than 250.

$\frac{1^2+2^2}{3}, \frac{2^2+3^2}{4}, \frac{3^2+4^2}{5}$ ______ -.-.-.-.-.-.-.-.-

```
#include<stdio.h>
void main(void)
{
        float term=0;
        int i=1;
        while(term<250)
        {
                term=((i*i)+(i+1)*(i+1))/((float)(i+2));
                printf("\t%f",term);
                i++;
        }
}
```

Example 5.19: Write a program to print the Fibonacci series until the term is less than 500. [A series Starting with 0 and 1, each new number in the series is simply the sum of the two before it. i.e., 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .]

```
#include<stdio.h>
void main()
{
        int i, firstterm=0,secondterm=1,  term=firstterm+secondterm;
        printf("%d %d",firstterm,secondterm);
        while(term<500)
        {
                printf(" %d",term);
                firstterm=secondterm;
                secondterm=term;
                term=firstterm+secondterm;
        }
}
```

Example 5.20: Write a program that will generate every third integer beginning with i=2 and continuing for all integers that are less than 100. Calculate the sum of these numbers that are exactly divisible by 7. [059/p/3-b]

```
#include<stdio.h>
void main(void)
{       int i,sum=0;
        for(i=2;i<100;i=i+2)
        {
                if(i%7= =0)
                {
                        sum=sum+i;
                        printf("\n%d",i);
                }
        }
        printf("\nThe sum of the numbers is =%d",sum);
}
```

Example 5.21: Write a program to read an integer number and add the individual digits contained in it until the final sum is a single digit. For example if entered number is n=2175698 then 2+1+7+5+6+9+8=38.
=>3+8 = 11
=>1+1 = 2 is the final answer. [2063/p/7]

```
#include<stdio.h>
void main()
{
        int sum,rem;long int n;
        printf("Enter an integr number:");
        scanf("%ld",&n);
        do
        {       sum=0;
                do
                {
                        rem=n%10;
                        sum=sum+rem;
                        n=n/10;
                }while(n!=0);
                n=sum;
        }while(sum/10!=0);
        printf("Sum=%d",sum);
}
```

Output

```
Enter an integr number: 2175698
Sum=2
```

Example 5.22: A program to find and print Armstrong numbers in range between two numbers given by a user. (Hint: 153 is Armstrong number because $1^3+2^3+3^3 = 153$)

```
#include<stdio.h>
void main(void)
{
        int m,n,r,i,k,sum;
        printf("Enter starting number:");
        scanf("%d",&n);
        printf("Enter ending number:");
        scanf("%d",&m);
        for(i=n;i<=m;i++)
        {    k=i;sum=0;
                do
                {
                        r=k%10;
                        sum=sum+r*r*r;
                        k=k/10;
                }while(k!=0);
                if(sum==i)
                        printf("\t%d",i);
        }
}
```

```
Enter starting number:1
Enter ending number:1000
1    153    370    371    407
```

Example 5.23: Write a program to print the following pattern of numbers.

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

```
#include<stdio.h>
void main(void)
{
        int i=1,j,k,nor,i_increment=1, k_increment;
        printf("Enter numbrs of rows:");
        scanf("%d",&nor);
        while(i>=1)
        {
                k_increment=1;
                k=1;
                for(j=nor-i;j>=1;j--)
                printf("\t ");
                while(k>=1)
                {
                        printf("\t%d",k);
                        if(k==i)
                        k_increment=-k_increment;
                        k=k+k_increment;
                }
                if(i==nor)
                i_increment=-i_increment;
                i=i+i_increment;
                printf("\n");
        }
}
```

Example 5.24: Write a program to evaluate the following series up n terms. Prompt the user to input value of n and x.
$f(x)=1- x^2/2!+x^4/4!-x^6/6!+x^8/8!$..........................
To evaluate any series, it is required to know its general term. The general term of the above series is
$(-1)^i x^{2i}/2i!$ where i=0,1,2,3..........

```
#include<stdio.h>
#include<math.h>
void main(void)
{
        int n,i,j,sign=-1,denominator;
        float sum=0,term,x,numerator;
        printf("Enter number of terms and value of x:");
        scanf("%d%f",&n,&x);
        for(i=0;i<=n-1;i++)
        {
                sign=sign*(-1);
                numerator=pow(x,2*i);
                denominator=1;
                for(j=1;j<=2*i;j++)
                denominator=denominator*j;
                term=sign*numerator/denominator;
                printf("term=%f\n",term);
                sum=sum+term;
        }
        printf("The sum of %d terms of the series=%f",n,sum);
}
```

Output

```
Enter number of terms and value of x: 4      1.5
term=1.000000
term=-1.125000
term=0.210938
term=-0.015820
The sum of 4 terms of the series=0.070117
```

Example 5.25: Write a program that reads numbers until the user enters a negative number. The Program should check all the numbers whether it is prime or not. If a number is Prime the program should display the number and at the last display the count of prime numbers entered. [059/P/8]

*In mathematics, a **prime number** (or **prime**) is a natural number greater than one whose only positive divisors are one and itself. A natural number that is greater than one and is not a prime is called a **composite number**. The numbers zero and one are neither prime nor composite.*

```
#include<stdio.h>
void main(void)
{
        int i,j,n,count=0;
        while(1)
        {       printf("Enter a number other than 0 and 1:");
                scanf("%d",&n);
                if(n<0)
                break;
                for(j=2;j<n;j++)
                {
```

```
            }
            if(n==j)
            {
                count++;
                printf("prime %d\n",n);
            }
            else
            printf("Not prime %d\n",n);
        }
        printf("Number of prime numbers are %d",count);
}
```

Example 5.26: Write a program to find the HCF of two integer numbers just entered by user. [2063/p/6]

```
#include<stdio.h>
void main(void)
{
        int firstnum, secondnum, remainder;
        printf("Input two integer numbers:");
        scanf("%d%d",&firstnum,&secondnum);
        do
        {
            remainder=firstnum%secondnum;
            if(remainder==0)
                printf("HCF=%d",secondnum);
            else
            {
                firstnum=secondnum;
                secondnum=remainder;
            }

        }while(remainder!=0);
}
```

Example 5.27: A program to find binary equivalent of a decmal integer number.

```
#include<stdio.h>
void main()
{
        int num,i,base=1,sum=0,rem;
        printf("Enter a number:");
        scanf("%d",&num);
        i=num;
        while(num!=0)
        {
            rem=num%2;
            sum=sum+rem*base;
            base=base*10;
            num=num/2;
        }
        printf("Binary equivalent of %d is %d",i,sum);
}
```

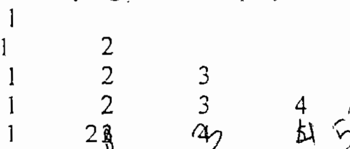
Output

```
Enter a number:13
Binary equivalent of 13 is 1101
```

Exercise 5

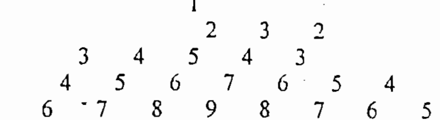
1. Explain the basic structures of a C programming.
2. Why is C called a structured programming language? Write about the control structures in C language.
3. Explain the different types of selective and repetitive statements with general form, flowchart and working process with examples.
4. What are the types of constants that can be used in a switch construct?
5. Differentiate between else...if ladder and switch statements.
6. Differentiate between while and do...while loop. .
7. Briefly discuss about three operational parts of for loop.
8. Differentiate between entry and exit controlled loop.
9. How can you distinguish loops as definite and indefinite?
10. What are the different variations of for loop?
11. Why do we need break, continue statement?
12. Differentiate between break and continue statement with examples and flowchart. [2063/b/4]
13. Why do you think the use of the goto statement is generally discouraged? Under what condition are they specially useful?
14. Can a test expression be other than the relational expression? Justify.
15. Write a program to print the numbers which are divisible by 3 but not by 5 from first 100 natural numbers.
16. Write a program to read the age of 100 persons and the count the number of persons in the age group 50 to 60. Use for and continue statements.
17. Write a program to print the factorial of number entered by the user. Use any loop of your choice. 7! =7*6*5*4*3*2*1
18. Write a program to count and print prime numbers between two numbers entered by a user.

19. Write a program to find the reverse of the number entered by the user. Then check whether the reverse is equal to the original number. If the reverse and original numbers are equal, display a message telling that the number is a palindrome; otherwise, not a palindrome.
20. Write a program that finds the LCM of two numbers. [Hint LCM=(product of two numbers)/HCF]
21. Using while loop only, write a program to print the following series until the term value is less than 750. The series is 1,2,5,10,17,26,37...............[2063/b/6]
22. Write programs to display the following patterns of numbers.



```
1
1   2
1   2   3
1   2   3   4
1   2   3   4   5
```

23. Write a program to print following type of pyramid of numbers. Try for inverse pyramid also.



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

24. Write separate programs to evaluate the following series.

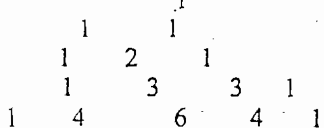
$S=1+(1+2)+(1+2+3)+(1+2+3+4)+(1+2+3+4+5)\ldots\ldots$

$1/(1-x)=1+x+x^2+x^3+\ldots\ldots+x^n$

$f(x) = 1+x^2/2!-x^4/4!+x^6/6!-x^8/8!\ldots\ldots$

$\sin x=x-x^3/3!+x^5/5!-x^7/7!+x^9/9!\ldots\ldots$

25. Write a program to generate a Pascal's triangle.



```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

Chapter 6
Functions

6.1 Introduction

A function is a self-contained program segment that carries out some specific, well-defined task. Every C program consists of one or more programs. Execution of program will always begin by carrying out the instructions in main. Additional functions will be subroutine to main. Functions are used to encapsulate a set of operations and return information to the main program or calling routine. Encapsulation is detail, information or data hiding. Once a function is written, we need only be concerned with what the function does. That is, what data it requires and what outputs it produces. The details, "how" the function works, need not be known. The use of functions provides several benefits. **First**, it makes programs significantly easier to understand and maintain. The main program can consist of a series of function calls rather than countless lines of code. A **second** benefit is that well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions. A **third** benefit of using functions is that different programmers working on one large project can divide the workload by writing different functions. **Fourth**, programs that use functions are easier to design, program, debug and maintain. Let us start from example 6.1.

Example 6.1 In this program function main calls a function named add. Function main passes two arguments and the function add returns the sum of the passed numbers to function main using return statement.

```
#include<stdio.h>
float add(int, float);          /* function declaration (function prototype) */
void main()
{
        int a;
        float b,sum;
        printf("Enter values of a and b:");
        scanf("%d%f",&a,&b);
        sum=add(a,b);     // function call, giving arguments a and b. where a and b are actual arguments(parameters)
        printf("\nThe sum=%f",sum);
}
float add(int p, float q)           /* function definition starts form this line, this first line is function declarator.
                                       Where p and q are formal arguments(parameters).*/

{                          /* function  body starts from this brace.*/
        float s;
        s=p+q;
        return s;          /* return statement with return statement s. */
}                          /* end of function definition and function body. */
```

Output

Enter values of a and b: 5 20
The sum=25.000000

6.2 Concept associated with functions

- Function declaration or prototype
- Function definition (function declarator and function body)
- Passing arguments
- Return statement
- Function call
- Combination of function declaration and function definition

6.2.1 Function declaration or prototype

A function declaration provides the following information to the compiler:

- The name of the function
- The type of the value returned (optional, default return type is integer).
- The number of arguments that must be supplied in a call to the function.
- The type of arguments that must be supplied in a call to the function.

When a function call is encountered, the compiler checks the function call with its declaration so that correct argument types are used.

Syntax: **return_type function_name(type1, type2, type3,.......,type4);**

In example 6.1, **float add(int, float);** is function prototype.

6.2.2 Function definition (function declarator and function body)

A function definition has two principal components:

a. Function declarator (The first line of the function definition) : The function declarator and the declaration must use the same function name, number of arguments, argument types and the return type but it does not have semicolon at the end.

Syntax- **return_type function_name(type1 argument1, type2 argument2, type3 argument3 ,.......,type n argument n);**

Note: where argument1, argument2, argument3....argument n are called formal arguments or formal parameters which are local to the function.

b. Body of the function: Function declarator is followed by the function body. It is composed of statements that make up the function, delimited by braces which defines the actions to be taken by the function.

6.2.3 Passing arguments

Providing values to the formal arguments of called function through the actual arguments of the calling function. (soon after , this topic will be discussed in detail)

Note: The function which calls other function is called the calling function and the function which is called by other function is called the called function .In example 6.1, main is a calling function because it calls the add function and add function is a called function which is called by the main function. Here add function can call other functions also. At that time the add function becomes a calling function.

6.2.4 Return statement

A function may or may not send back any value to the calling function. If it does, it is through the **return** statement. While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most. The return statement can take one of the following forms

return; or **return(expression)**

the plain return does not return any value; it acts as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

```
Example: if(error)
             return;
```

The second form of return expression returns the value of the expression. In the above example, return expression **returns the value of s**. It is possible for a function to have multiple return statements. Following set of statement shows the cocncept.

```
int multret(char command)
{
    switch (command)
    {
                    case'+': return 1 ;
                    case'-': return 2;
                    case '*': return 3;
                    case '/': return 4;
                    default : return 0;
    }
}
```

6.2.5 Function call

A function is a inactive part of the program which comes in to life when a call is made to the function. A function call is specified by the function name followed by the values of parameters enclosed within parentheses, terminated by a semicolon. In the above example, add(a,b). is the function call and a and b are the parameters. *Note: the number, type and order of arguments in the function declaration, function call and function declarator must be the same.*

6.2.6 Combination of function declaration and function definition

If the functions are defined before they are called , then the declaration is unnecessary. See example 6.2. Where the function add is defined before main i,e, it is defined before calling it.

Example 6.2: This example shows the combination of function declaration and definition.

```
#include<stdio.h>
float add(int p, float q)
{
    float s;
    s=p+q;
    return s;
}
void main()
{   int a;
    float b,sum;
    printf("Enter values of a and b:");
    scanf("%d%f",&a,&b);
    sum=add(a,b);
    printf("\nThe sum=%f",sum);
}
```

6.3 Categories of functions

- Library functions
- User defined functions

6.3.1 Library functions

These functions have already been written, compiled and placed in libraries. printf , scanf , getch etc are examples of library functions.

6.3.2 User defined functions

These are the functions which can be defined and used by the programmers in C programs according to their requirements.

6.4 Categories of user defined functions

Depending on whether arguments are present or not and whether a value is returned or not, user defined functions are categorized as follows:

6.4.1 Functions with arguments and return value

The function add in example 6.1 falls in this category. It takes two arguments a and b from the calling function (main) and it returns the sum to the calling function.

6.4.2 Functions with arguments and no return value

Example 6.1 can be written in the following form where the called function takes the arguments (a and b) from the calling function but do not return the sum(s) to the calling function. The function add displays the result itself. So return type is void and the **return** statement is not required in the called function.

Example 6.3: This example shows the separation of function declaration and definition.

```
#include<stdio.h>
void add(int, float);
void main()
{
        int a;
        float b;
        printf("Enter values of a and b:");
        scanf("%d%f",&a,&b);
        add(a,b);
}
void add(int p, float q)
{       float s;
        s=p+q;
         printf("\nSum=%f",s);
}
```

6.4.3 Functions with no arguments and no return value

The objective of Example 6.3 can also be achieved as in example 6.4. Study the following example and find the differences.(void return type, no arguments, no return statement, function add reads the value of a and b itself and displays the result(s) from the function body, main function only calls the add function and all the other activities are done inside the called function body.)

Example 6.4: This example shows the technique to write functions with no arguments and no return values.

```
#include<stdio.h>
void add();
void main()
{    add();
}
void add()
{
    int a; float b,s;
     printf("Enter values of a and b:");
     scanf("%d%f",&a,&b);
     s=a+b;
     printf("\nThe sum=%f",s);
}
```

6.4.4 Functions with no arguments and return value

Study the following example and find the differences.

Example 6.5 This example shows the technique to write functions with no arguments and return values.

```
#include<stdio.h>
float add();
void main()
{   float x;
     x=add();
     printf("The sum is %f",x);
}
float add()
{    int a;
     float b;
     printf("Enter values of a and b:");
```

```
    scanf("%d%f",&a,&b);
    return a+b; /* we can write a expression in the return statement. */
}
```

Example 6.6 A program having more then one user defined functions.

```
#include<stdio.h>
float add(float, float);
float sub(float,float);
float mul(float,float);
float div(float,float);
void main()
{
    float a,b,sum,dif,product,quotient;
    char ch,flag;
    do{
        printf("\n Enter any operator(+,-,*,/):");
        scanf(" %c",&ch);
        printf("\n Enter values of a and b:");
        scanf("%f%f",&a,&b);
        switch(ch)
        {
            case '+':
            sum=add(a,b);
            printf("sum=%f",sum);
            break;
            case '-':
            dif=sub(a,b);
            printf("Difference=%f",dif);
            break;
            case '*':
            product=mul(a,b);
            printf("Product=%f",product);
            break;
            case '/':
            quotient=div(a,b);
            printf("Quotient=%f",quotient);
            break;
            default:
            printf("Enter any above shown characters");
            break;
        }
        printf("\n\nEnter e to exit\nany character for continue\t");
        scanf(" %c",&flag);
    }while(flag!='e');
}
float add(float p, float q)
{
    return p+q;
}
float sub(float p, float q)
{
    return p-q;
}
float mul(float p, float q)
{
    return p*q ;
}
float div(float p, float q)
{
    return p/q;
}
```

In example 6.6, there are 4 user defined functions for addition, subtraction, multiplication and division of two numbers. These functions are defined after the definition of main function. This program at first asks the user to input the operator according to the user requirement. Then asks for two operands to be operated by the just given operator. The operator is used as the switch expression. When the switch expression is matched to a case label, then the corresponding function is called to perform the operation. The calling (main) function passes two arguments to the called function. The called(add, sub , mul, div) function performs the required operation on the arguments and returns the result to calling (main) function. This program runs continuously until the user enters e to exit i.e, program is running in do…while loop.

Sample output

```
Enter any operator(+,-,*,/): *
Enter value of a and b: 5   6
Product=30
```

6.5 Ways of passing arguments to functions

There are two different mechanisms to pass arguments to a function.

- Passing by value
- Passing by reference

6.5.1 Passing by value

The process of passing the actual value of variables is known as passing by value (call by value). In this mechanism, the values of the actual arguments are copied to the formal arguments of the function definition. So, the value of the arguments in the calling function are not changed even they are changed in the called function. In above-illustrated examples, we have passed the arguments by value. It does not allow information to be transferred back to the calling function via the arguments. Thus passing values to functions is restricted to a one-way transfer of information. In the example 6.7, values of x and y are passed to p and q. Inside the function values of p and q are swapped but that was not reflected to the allocated memory location of actual variables x and y.

Example 6.7: Write a program to swap the values of two variables using a function.

```
#include<stdio.h>
void swap(int, int );
void main()
{       int x,y;
        printf("Enter x and y:");
        scanf("%d%d",&x,&y);
        printf("before swap \nx=%d\ny=%d",x,y);
        swap(x,y);
        printf("\n\nafter swap \nx=%d\ny=%d",x,y);
}
void swap(int p, int q)
{
        int temp;
        temp=p;
        p=q;
        q=temp;
}
```

Output

```
Enter x and y:8 9
before swap  x=8  y=9
after swap x=8  y=9
```

In example 6.7, values of two operands are passed from main to the swap function. Swap function swaps the values of the local variables (p and q) which are used to copy the values sent by the main. But it is required to swap the values of x and y. In passing by value method, any change on p does not reflect to x. Similarly, q to y. Return statement can not return more than one values at a time. This is the main disadvantage of passing by value method. The main advantage of this method is that the original value of the variable remains the same after the function call. The sample output shows the values of x and y. which are same before and after function call. How to swap the values of x and y? It can be done using passing by reference. See example 6.8

6.5.2 Passing by reference

The process of calling a function using pointers to pass the address of variable is known as passing by reference (call by reference). In this mechanism, see examples 6.8, instead of passing x and y, we must pass the address of x and y. To receive the address, we must use pointer variables in the function declaration and definition instead of using normal variables. The function called by reference can change the value of the variable used in the call. In example 6.8, values of x and y are changed after a function call.

Example 6.8: Write a program to swap values of two variables using a function. Pass the arguments to function by reference.

```
#include<stdio.h>
void swap(int *,int *);
void main()
{       int x,y;
        printf("Enter x and y:");
        scanf("%d%d",&x,&y);
        printf("before swap \nx=%d\ny=%d",x,y);
        swap(&x,&y);
        printf("\n\nafter swap \nx=%d\ny=%d",x,y);
}
void swap(int *p, int *q)
{       int temp;
        temp=*p;
        *p=*q;
        *q=temp;
}
```

Output

```
Enter x and y:8 9
before swap  x=8  y=9
after swap x=9  y=8
```

In example 6.8, address of the variables is passed to the called function. Here addresses of x and y are passed to the function swap. These address are copied to p and q. Any operation done inside the swap is actually done on the memory location pointed by the address. Thus, swap done on *p and *p means swap done on x and y. Thus the effect is reflected to the calling function (main) though operations are done inside called(swap) function. The advantage of this method is more than one values can be returned from called function to calling function. The main disadvantage of this method is the original value of the passed variables does not remain the same after the function call.

6.6 Function main

Main is also a user defined function except its name, number and type of arguments. It is starting point of program execution. Each program must have a main function. Programmers can write statements in body of main to solve any problems. More complex problems can be divided into small easier chunks. Each small problem can be solved by writing other small functions separately. These all functions are required to be called from the main to combine the solutions and to get the integrated solution of the complex problem. The required statements must be written either inside the main or can be called to other function from the main. The general form of main is

```
main()
{ }
```

The above statement is sufficient for the execution of the program, though the program does not do any useful operation.

6.7 Recursion

Expressing an entity in terms of itself is called recursion. Recursion is a programming method in which a function calls itself.

6.7.1 Recursive function

A recursive function is a function that calls on itself. Two important conditions that must be satisfied by any recursive function are:

- Each time a function calls itself and it must be closer to a solution.
- There must be decision criteria for stopping the process.

When a recursive function is called, the function knows to solve only the simplest case.

```
recursive algorithm:
    if this is the simplest case
        solve it
    else
        redefine (simplify) the problem using recursion
```

To write a recursive function, we must be able to break the problem into subparts, at least one of which is similar in form to the original problem. There are two distinct parts to the function: The base case- This part does not call itself. It handles a simple case that we know how to do without breaking the problem down into a simpler problem. The recursive case - This part breaks the problem into a simpler version of the original problem. This part makes (at least) one recursive call to itself.

Basic steps of recursive programs

Every recursive program follows the same basic sequence of steps:

- Define base and recursive case(s).
- Initialize the algorithm. Recursive programs often need a seed value to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is non recursive but that sets up the seed values for the recursive calculation.
- Check to see whether the current value(s) being processed match the base case. If so, process and return the value.
- Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
- Run the algorithm on the sub-problem.
- Combine the results in the formulation of the answer.
- Return the results.

Example 6.9: Factorial of a number-The classic example of recursive programming involves computing factorials. The factorial of a number is computed as that number times all of the numbers below it up to and including 1. For example, factorial(5) is the same as 5*4*3*2*1, and factorial(3) is 3*2*1. An interesting property of a factorial is that the factorial of a number is equal to the starting number multiplied by the factorial of the number immediately below it. For example, factorial(5) is the same as 5 * factorial(4). You could almost write the factorial function simply as this:

```
int factorial(int n)
{
        return n * factorial(n - 1)
}
```

The problem with this function, however, is that it would run forever because there is no place where it stops. The function would continually call factorial. There is nothing to stop it when it hits zero, so it would continue calling factorial on zero and the negative numbers. Therefore, our function needs a condition to tell it when to stop. Since factorials of numbers less than 1 don't make any sense, we stop at the number 1 and return the factorial of 1 (which is 1). Therefore, the real factorial function will look like in the following program (**in bold**): [057/c/4-b]

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

```
        return 1;
        else
        return(n*factorial(n-1));
}
```

As we can see, the base case is 1 (here, in case of factorial base case can be 0 because factorial of 0 is also 1). The stopping point is called the base case. A base case is the bottom point of a recursive program where the operation is so trivial as to be able to return an answer directly. All recursive programs must have at least one base case and must guarantee that they will hit one eventually; otherwise the program would run forever or until the program run out of memory or stack space.

Let's trace the evaluation of

```
factorial(4):
    factorial(4) =
    4 * factorial(3) =
    4 * (3 * factorial(2)) =
    4 * (3 * (2 * factorial (1))) =
    4 * (3 * (2 * 1) =
    4 * (3 * 2) =
    4 * 6 =
    24
```

Example 6.10: Fibonacci Sequence [2063/p/8]

There is a famous sequence of numbers called the Fibonacci sequence. The Fibonacci sequence is $F(n) = F(n-1) + F(n-2)$, where $F(0) = 1$ and $F(1) = 1$. The Fibonacci series is a sequence of numbers in which each number is the sum of the previous two numbers. The first and second Fibonacci numbers are 0 and 1.

```
#include<stdio.h>
void fibonacii(unsigned int, unsigned int, unsigned int);
void main()
{
        unsigned int n;
        printf("Enter number of terms:");
        scanf("%u",&n);
        fibonacii(0,1,n);
}
void fibonacii(unsigned int first, unsigned int second, unsigned int noft)
{
        if(noft!=0)
        {
                printf("%u\t",first);
                fibonacii(second,second+first,noft-1);
        }
}
```

Output

```
Enter number of terms: 10
0  1  1  2  3  5  8  13  21  34
```

Example 6.11: The greatest common divisor (Highest common factor)

The greatest common divisor of integers x and y is the largest integer that evenly divides both x and y. Write a recursive function gcdhcf that returns the greatest common divisor of x and y, which is defined recursively as follows:

If y is equal to 0, then gcdhcf(x,y) is x, otherwise gcdhcf(x,y) is gcdhcf(y, x % y).

Which is implemented in following program.

```
#include<stdio.h>
int gdchcf(int,int);
void main()
{       int a,b,gdc;
        printf("Enter two numbers to calculate gdc(hcf):");
        scanf("%d%d",&a,&b);
        gdc=gdchcf(a,b);
        printf("Greatest common divisor is %d",gdc);
}
int gdchcf(int x,int y)
{       if(y!=0)
        return gdchcf(y,x%y);
        else
        return x;
}
```

Example 6.12: Write a computer program to display a line of text "Programming is fun" 5 times using recursive function.[2058/P/3-b]

```
#include<stdio.h>
void main()
{       static int i=5;
        if(i>=1)
        {               i--;
                        printf("\nProgramming is FUN.");
```

```
    }
}
```

6.7.2 Comparing loops with recursive functions

Properties	Loops	Recursive functions
Repetition	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by either finishing the block of code or issuing a continue command.	Execute the same block of code repeatedly to obtain the result; signal their intent to repeat by calling themselves.
Terminating conditions	In order to guarantee that it will terminate, a loop must have one or more conditions that cause it to terminate and it must be guaranteed at some point to hit one of these conditions.	In order to guarantee that it will terminate, a recursive function requires a base case that causes the function to stop recurring.
State	Current state is updated as the loop progresses.	Current state is passed as parameter.

6.7.3 Advantages of Recursion

- The code may be much easier to write.
- To solve Some problems which are naturally recursive such as towers of Hanoi.

6.7.4 Disadvantages of Recursion

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on system stack so excessive recursion may overflow the system stack. (see chapter 7.3.1)
- It is difficult to think recursively so one must be very careful when writing recursive functions.

6.6 Scope of variables (Local and global variables)

Scope of a variable determines over what part(s) of a program a variable is actually available for use (active). On the basis of place of declaration, variables are categorized as local (internal, automatic) and global (external)

6.8.1 Local

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Example 6.13: This example illustrates the concept of local variables.

```
#include<stdio.h>
void function1();
void function2();
void main()
{           int i=5000;
            function2();
            printf("%d\n",i);
}
void function1()
{           int i=50;
            printf("%d\n",i);
}
void function2()
{           int i=500;
            function1();
            printf("%d\n",i);
}
```

Output

```
50
500
5000
```

In example 6.13, i is declared inside there functions. Although the name of variables is same, they are treated as different variables because memory allocation is done separately. One function does not know the variables inside the other functions. For example, here function main does not know the i's inside function function1 and funcion2. Here, main calls function2, function2 again calls function1 so the value of i in function1 is displayed i.e., 50. Then control goes back to function2 that displays the value of its i i.e., 500. After that, control returns back to main and prints the value of i inside main i.e., 5000.

6.8.2 Global

These variables can be accessed by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. To declare a global variable, declare it outside of all the functions. There is no general rule for where outside the functions these should be declared, but declaring them on top of the code is normally recommended for reasons of scope, as explained below. If a variable of the same name is declared both within a function and outside of it, the function will use the variable that was declared within it and ignore the global one. But it is recommend that to use as few global variables as possible.

Example 6.14: This example illustrates the concept of gloabal variables.

```
#include<stdio.h>
void function1();
void function2();
int a,b,c;
void main()
{       function1();
        a=10,b=20;
        c=a+b;
        printf("in main a+b=%d\n",c);
}
void function1()
{       function2();
        printf("in function1 a+b=%d\n",a+b);
}
void function2()
{       printf("Enter value of a and b:");
        scanf("%d%d",&a,&b);
}
```

Output

```
Enter value of a and b:3 4
in function1 a+b=7
in main a+b=30
```

In example 6.14, a,b and c are global variables because they are declared above the function main. Variables a, b and c are visible in all functions. They can be accessed equally from all the functions.

6.7 Extent of variables(life time, longevity)

The time period during which memory is associated with a variable is called the extent of the variable. The life time of a variable is categorized by its storage class. The storage class of a variable indicates the allocation of storage space to the variable. The storage class are of four types.

- **auto variables**
- **register variables**
- **static variables**
- **extern variables**

6.9.1 auto variables

All variables declared within a functions are auto by default. The extent of a variable is defined by its scope. Variables declared auto can only be accessed within the function or nested block within which they are declared. They are created when the control is entered into and destroyed when the control is exited. They can be declared as : **auto** int a;

6.9.2 register variables

C allows the use of the prefix **register** in primitive variable declarations. Such variables are register variables. They are stored in the **registers** of the microprocessor. The number of variables, which can be declared register, are limited. A program that uses register variables executes faster as compared to the similar program without register variables. Loop indices, accessed more frequently, can be declared as register variables.

Example 6.15: This program shows the declaration of register variables.

```
#include<stdio.h>
#include<string.h>
void main()
{       register int i;
        char name[100];
        printf("Enter a string:");
        gets(name);
        printf("The reverse of the string is:");
        for(i=strlen(name)-1;i>=0;i--)
        printf("%c",name[i]);
}
```

Output : Enter a string: kathmandu engineering college
The reverse of the string is: egelloc gnireenigne udnamhtak

6.9.3 Static variables

As the name suggests, the value of the static variables persists until the end of the program. A variable can be declared static using the keyword static like **static** int p; Static variable may be internal or external type, depending on the place of declaration.

Example 6.16: This program shows the declaration of register variables.

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

In example 6.16 p is static variable. It retains its previous value so that the old value is incremented by 1 in each function call. Which is shown in the output.

Output

```
p=6
p=7
p=8
p=9
```

6.9.4 extern variables

Up to now, all the examples have shown an entire program within one file. This is fine for short and simple programs but any large real world program is likely to have its code distributed among **multiple files**. There are several practical reasons for organizing a program into multiple files. First, it allows parts of the program to be developed independently, possibly by different programmers. This separation also allows independent compiling and testing of the modules in each file. Second, it allows greater reuse of code. Files containing related functions can become libraries of routines that can be used in multiple programs. Having a program distributed in multiple files raises global or external variables. Within one file, the scope of a global (external) variable is from its definition to the end of the file. The keyword **extern** makes it possible to use a global variable in multiple files. Notice that a variable may only be defined once. It may be declared multiple times. Remember that a definition creates space or reserves memory, for the variable. The declaration just informs the compiler that a variable or function exists. The keyword extern tells the compiler that the variable is **defined** in another file. In the following example g is declared and defined in mfile1 and it is declared as extern in mfile2.

Example 6.17 : This example illustrates the concept of multifile programming and extern type variables.

```
/*mfile1.c*/
#include<stdio.h>
int g;
void function1();
void function2();
void function3();
void main()
{           int i;
            printf("Enter the value of g:");
            scanf("%d",&g);
            function1();
            function2();
            function3();
}
void function1()
{
            printf("\nIn MFILE1 and function1 %d",g+1);
}
/* mfile2.c */
#include<stdio.h>
extern int g;
void function2()
{
            printf("\nIn MFILE2 and function2 %d",g+2);
}
void function3()
{
            printf("\nIn MFILE2 and function3 %d",g+3);
}
```

In example 6.17, there are two C files: **mfile1.c and mfile2.c**. main function is in mfile1.c. Some required functions are declared in mfile1.c and defined in mfile2.c. A variable g is declared globally in mfile1.c. In mfile2.c, g is declared as extern. It means g is declared externally somewhere outside the mfile2.c

Output

Enter a value of g:20

In MFILE2 and function2 22.
In MFILE2 and function3 23

Table 6.2 summary of storage classes

Storage class	storage	Default Initial value	Life time
auto	memory	garbage	Till the control remains within the block in which the variable is defined if it is defined globally it is visible to all the functions.
static	memory	zero	Value of the variable persists between different function calls
register	register	garbage	Till the control remains within the block in which the variable is defined if it is defined globally it is visible to all the functions
extern	memory	zero	As long as the program execution does not come to the end.

6.10 Some important examples

Example 6.18: Write a program to evaluate the sine series.
$Sin(x)=x-x^3/3!+x^5/5!-x^7/7!+x^9/9!$................

We must find the general term before writing program. The general term of this series is
$(-1)^{n-1}\ x^{(2n-1)}/(2n-1)!$,n=1,2,3,4,.......

```
#include<stdio.h>
#include<math.h>
long int fact(int j)
{
        if(j==1)
        return 1;
        else
        return(j*fact(j-1));
}
float numerator(float x, int n)
{       float product=1,i;
        for(i=1;i<=n;i++)
        product=product*x;
        return product;
}
void main(void)
{
        int n,i,j,sign,power;
        float term,x,sum=0,numerat;
        long int denum;
        printf("\nEnter numbers of terms:");
        scanf("%d",&n);
        printf("\nEnter the value of angle in Degree:");
        scanf("%f",&x);
        x=x*3.14/180; /* to convert degree to radian. If value of x is in radian , it is not required. */
        for(i=1;i<=n;i++)
        {               sign=pow(-1,i-1);
                        power=2*i-1;
                        numerat=numerator(x,power);
                        denum=fact(power);
                        printf("\nfactorial of %d is:%ld" ,power, denum);
                        term=sign*numerat/denum; /*we can use pow(x,power)*/
                        printf("\nTerm:%f",term);
                        sum=sum+term;
                        printf("\n");
        }
        printf("\nSum=%f",sum);
}
```

Output

Enter numbers of terms:8	Term:-0.004665
Enter the value of angle in Degree: 90	factorial of 9 is:362880
factorial of 1 is:1	Term:0.000160
Term:1.570000	factorial of 11 is:39916800
factorial of 3 is:6	Term:-0.000004
Term:-0.644982	factorial of 13 is:1932053504
factorial of 5 is:120	Term:0.000000
Term:0.079491	factorial of 15 is:2004310016
factorial of 7 is:5040	Term:-0.000000
	Sum=0.999999

$sn=1+x/1!+x^2/2!+x^3/3!$.................... [062/b/3-b]

```
#include<stdio.h>
#include<math.h>
long int fact(int);
void main()
{
        float x;
        double term=0,sum=0;
        int i;
        clrscr();
        printf("Enter value of x:");
        scanf("%f",&x);
        i=0;
        do
        {
                sum=sum+term;
                term=pow(x,i)/fact(i);
                printf("term=%lf\n",term);
                i++;
        }while(term>0.000001);
        printf("Sum=%lf",sum);
        getch();
}
long int fact(int n)
{
        long int f=1;int j;
        if(n==0)
        return 1;
        else
        {
        for(j=1;j<=n;j++)
        f=f*j;
        return f;
        }
}
```

Output

```
Enter value of x:1.45
term=1.000000
term=1.450000
term=1.051250
term=0.508104
term=0.184188
term=0.053414
term=0.012908
term=0.002674
term=0.000485
term=0.000078
term=0.000011
term=0.000001
term=0.000000
Sum=4.263115
```

Example 6.20: Write a recursive function to find the sum of natural numbers. Test the function in your main program. [062/p/8]

In mathematics, a **natural number** is either a positive integer (1, 2, 3, 4, ...) or a non-negative integer (0, 1, 2, 3, 4, ...). The former definition is generally used in number theory, while the latter is preferred in set theory.

```
#include<stdio.h>
void main()
{
        static int i=1,sum=0,n;
        if(i==1)
        {
                printf("Enter a number:");
                scanf("%d",&n);
        }
        if(i<=n)
        {
                sum=sum+i;
                i++;
                main();
        }
        else
        {
                printf("Sum=%d",sum);
        }
}
```

Example 6.21: Write a function that takes an integer number of any digits and returns the number reversed. In the calling function read a number, pass it to the function, and display the result. [061/p/4-b]

```
#include<stdio.h>
int rev(int);
void main()
{           int n;
            clrscr();
            printf("Enter a integer number:");
            scanf("%d",&n);
            printf("Reversed number=%d",rev(n));
            getch();
}
int rev(int num)
{
            int reverse=0,remainder;
            while(num!=0)
            {
                        remainder=num%10;
                        reverse=reverse*10+remainder;
                        num=num/10;
            }
            return(reverse);
}
```

Output

Enter a integer number:12341
Reversed number=14321

Above problem can be asked to check whether the entered number is palindrome or not. If the reversed number is equal to the given number, then the number is palindrome.

Example 6.22: Write a program to read an integer number, count the even digits and odd digits present in the entered number. You must write a function for calculating the result and the result must be displayed in main function itself.

```
#include<stdio.h>
void odd_even_digit_count(int,int*,int*);
void main()
{
            int num,odd_count=0,even_count=0;
            printf("Enter an integer number:");
            scanf("%d",&num);
            odd_even_digit_count(num,&odd_count,&even_count);
            printf("Number of odd numbers=%d",odd_count);
            printf("\nNumber of even numbers=%d",even_count);
}
void odd_even_digit_count(int n,int *oc,int *ec)
{
            int rem;
            while(n!=0)
            {
                        rem=n%10;
                        if(rem%2==0)
                                    *ec=*ec+1;
                        else
                                    *oc=*oc+1;
                        n=n/10;
            }
}
```

Output

Enter an positive integer number:12345
Number of odd numbers=3
Number of even numbers=2

Exercise 6

1. What is a function? What are its advantages? [059/p/5-a]
2. Differentiate library and user defined functions.
3. What are different types of user defined functions? Explain with examples.
4. Write short notes on "Function Prototype". [058/P/4-a]
5. What is a function declarator?
6. Differentiate between function declaration and definition.
7. What do you understand by return statement? Explain with suitable examples about its advantages and disadvantages. [061/b/4-a]
8. What is the main limitation of return statement and how can we overcome this? [060/P/]
9. Differentiate between the methods of passing arguments to a function with examples. What are their advantages and disadvantages? [060/p]
10. What is a recursive function? What are its requirements? Explain with an example. [059/C/4-a]
11. Compare loops with recursive function.

12. What are advantages and disadvantages of recursion?.
13. What do you mean by storage class of variables? Write down about static, local, and global variables in terms of lifetime and scope with example. [062/b]
14. Classify the variables according to the scope and extent (storage class) with examples. [2063/p/3]
15. Write a program to print the characters from 0 to 255 using do...while loop in the functions.
16. Write a complete program that reads numerical values for x and n and evaluate the formula $y=x^n$. Your program must have a function to calculate the result and return it. Assume both x and n are integer and you can not use pow() function. [058/c/3-b]
17. Write a program that takes two integers, find their highest common factor and Lowest common multiple and displays them. The highest common factor must be computed in a function called hcfactor. Use both recursive and non-recursive methods. [059/p/5/b]
18. Write a program, which asks two operands and a operator, passes them to a function, performs the corresponding operator and returns the result to the calling function.
19. Write a program to calculate to calculate sum of digits in an integer number using function.
20. Write a program to evaluate the following series to n terms specified by the user. Here the factorial must be calculated by recursion $x/1!-x^3/3!+x^5/5!$................. [061/b/5-b]
21. Write a program to convert a character from lower to upper or upper to lower case using Call by Referece method.
22. Write a program to find the largest number among four numbers using a function that returns the largest of three numbers passed to it.
23. Write a program to read an integer number, count the even digits and odd digits present in the entered number. You must write a function for calculating the result and the result must be displayed in main function itself. [2063/b/8]
24. Write a program to display all the prime numbers within the ranges specified by the users using functions. [2063/b/7]
25. Write a program that will have four user-defined functions. They will do following different tasks:
    - Calculation of simple interest.
    - Finding prime numbers in the range between n1 and n2.
    - Finding the roots of a quadratic equation.
    - Finding the three digits Armstrong numbers.

Before calling above functions, ask the user to choose any operation then call the corresponding function using switch function. Declare the required variables in the corresponding function and display the result from the same function.

Chapter 7

7.1 Arrays 7.2 Pointers

7.3 Dynamic Memory Allocation 7.4 Strings

7.1.1 Introduction

An array is a group of related data items that share a common name. Arrays are derived data types. Arrays are also classified as aggregate data types. Unlike atomic data types which cannot be subdivided, aggregate data types are composed of one or more elements of an atomic or another aggregate data type. Simply aggregate data types can be subdivided or broken down. Arrays can be of any data type supported by C, including constructed data types. Arrays can be of any storage class except 'register'. The use of arrays allows for the development of smaller and more clearly readable programs.

7.1.2 Types of arrays

7.1.2.1 One-dimensional arrays

Elements of the array can be represented either as a single column or as a single row. While dealing with one-dimensional arrays, we specify a single subscript to locate a specific element.

7.1.2.1.1 Declaring one-dimensional array

Arrays can be declared as local variables or global variables.

Global declaration

```
char string[50];  /* Global declaration only  */
void main()
{
    .
    .
}
```

Local declaration

```
void main()
{
        float marks[20];  /* Local declaration only */
        .
        .
}
```

7.1.2.1.2 Initializing one-dimensional array

we can initialize the elements of array in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

data-type name[size]={list of arguments};

To initialize an array when it is declared does not require to give the size of the dimension in the array declaration.

Examples:

Global initialization

```
char name[] = "John Smith";  /* Global  initialization */
int age[] = {21,32,43,22,25};  /* Global  initialization */
main()
{
    .
    .
}
```

Local initialization

```
void main()
{
        float marks[20];  /* Local declaration only */
        int age[] = {21,32,43,22,25};  /* Local declaration and initialization */
    .
    .
}
```

Initialization of array in C suffers two drawbacks:

- There is no convenient way to initialize only selected elements.
- There is no shortcut method for initializing a large numbers of array elements.

7.1.3 Accessing array elements

To access individual elements of an array, a subscript or index must be used. The subscript identifies the sub-element or component of the array that is to be accessed. Subscripts must start at 0 and proceed in increments of 1. To know how array elements are stored in the memory, study the following figure. Where marks is an array of type float of size n.

address	1000	1004	1008	1012		1000+(n-1)*4
marks	90	80	76	45	------	85
elements	marks[0]	marks[1]	marks[2]	marks[3]		marks[n-1]

Note: marks[0] is 90 which is stored at 1000.
Never forget, marks holds the address of marks[0]

Fig. 7.1.1 Memory allocation of array marks

If an array is declared to hold ten elements, the subscripts start at 0 and the highest subscript allowed for that array is 9. There is no **bound checking** on arrays. It is the programmer's responsibility to keep the subscript within the bounds of the array. On the above array **marks**, which has n elements, if a statement such as marks[n] = 10 is made in the program, no error or warning will be generated by the compiler. The array **marks** only support subscripts ranging from **0** to **n-1**. A subscript of **n** is not within the bounds of the array, but C allows the statement. In most cases the program will be terminated with a runtime error because of the above statement.

Example 7.1.1: An example to illustrate how to input/output of array elements.

```
#include<stdio.h>
void main()
{
        float marks[5]; /* Array decleration */
        float age[]={23,19,20,21}; /* Array initialization */
        int i;
        for(i=0;i<5;i++)
        {
                printf("Enter element %d ",i);
                scanf("%f",&marks[i]); /* Array element input */
        }
        for(i=0;i<5;i++)
        {
                printf("%f ",marks[i]); /* Array element output */
        }
        printf("The initialized array is:\n");
        for(i=0;i<4;i++)
        {
                printf("%f ",age[i]); /* Initialized array element output */
        }
}
```

Output

```
Enter element 0  12
Enter element 1  34
Enter element 2  24
Enter element 3  45
Enter element 4  56
12.000000 34.000000 24.000000 45.000000 56.000000
The initialized array is:
23.000000 19.000000 20.000000 21.000000
```

Example 7.1.2: A program to find the average of an array elements.

```
#include<stdio.h>
void main()
{
        float marks[5],sum=0,avg;
        int i;
        for(i=0;i<5;i++)
        {
                printf("Enter element %d ",i);
                scanf("%f",&marks[i]);
                sum=sum+marks[i];
        }
        avg=sum/i; /* because latest value of i is 5. */
        printf("Average=%f",avg);
}
```

Output
Enter element 0 1
Enter element 1 2
Enter element 2 3
Enter element 3 4
Enter element 4 5
Average=3.000000

Example 7.1.3: A program to find the largest element of an array.

```
#include<stdio.h>
void main()
{
        float marks[5],max;
        int i;
        for(i=0;i<5;i++)
        {
                printf("Enter element %d ",i);
                scanf("%f",&marks[i]);

        }
        max=marks[0];
        for(i=1;i<5;i++)
        {
              if(marks[i]>max)
              {
                        max=marks[i];
                }
        }
        printf("\nThe largest element is %f",max);
}
```

Output
Enter element 0 34
Enter element 1 2
Enter element 2 45
Enter element 3 6767
Enter element 4 45
The highest element is 6767.000000
Similarly you can find the smallest element.

7.1.4 Sorting of an array elements

Ordering of data either in ascending or descending order is called sorting. Sorting is illustrated in the following example.

Example 7.1.4: Sorting of array elements in ascending order.

```
#include<stdio.h>
void main()
{
        int marks[5],temp,i,j;
        for(i=0;i<5;i++)
        {
                printf("Enter element %d ",i);
                scanf("%d",&marks[i]);
        }
        for(i=0;i<5-1;i++)
        {
          for(j=0;j<5-i-1;j++)
          {
                if(marks[j]>marks[j+1])
                {       temp=marks[j];
                        marks[j]=marks[j+1];
                        marks[j+1]=temp;
                }
          }
        }
        printf("Sorted array is:");
        for(i=0;i<5;i++)
        {
                printf("%d ",marks[i]);
        }
}
```

Oputput
Enter element 0 90
Enter element 1 89
Enter element 2 67
Enter element 3 45
Enter element 4 23

Sorted array is:23 45 67 89 90
This sorting technique is **bubble sort.** Similarly, we can do in descending order.

7.1.5 Searching

Finding the required elements from an array.

Example 7.1.5: Write a program to count and find the sum of the all numbers in the array which are exactly divisible by 7 but not by 5. [2062/b/2-b]

```
#include<stdio.h>
#include<math.h>
void main()
{
        int array[5],count=0,sum=0,i;
        for(i=0;i<5;i++)
        {
                printf("Enter array[%d]:",i);
                scanf("%d",&array[i]);
        }
        for(i=0;i<5;i++)
        {
                if(array[i]%7==0 && array[i]%5!=0)
                {
                        sum=sum+array[i];
                        count++;
                        printf("%d is requird array element.",array[i]);
                }
        }
        printf("\n Sum=%d",sum);
        printf("Total number of requird numbers is %d",count);
}
```

Output

```
Enter array[0]:35
Enter array[1]:49
Enter array[2]:56
Enter array[3]:7
Enter array[4]:70
49 is requird array element.
56 is requird array element.
7 is requird array element.
Total sum=112
Total number of requird numbers is 3
```

Sorting and searching are called operations on arrays.

7.1.6.6. One-dimensional arrays and functions

An array is a collection of contiguous storage locations. Each location holds one element. All elements must be of the same type. The array name by itself references the address of the beginning of the array. The array name is seldom used by itself, since we usually process arrays element by element. The array name by itself is primarily used to pass the array to a function. Since they are generally large structures, we do not want to waste time and space copying arrays when we pass them to subroutines. In C, arrays are automatically passed by reference to a function. The name of an array stores the beginning address of where the array data is stored in memory. In the above example of marks array, the name of the array marks holds the address of element marks[0] i.e marks=&mark[0]. When we pass an array by reference, the subroutine (function) has access to all the elements stored in the array and any change in the function will be reflected to the calling function. From above discussion, we conclude that an array can be passed to a function by passing only the array name and the size of the array. In the above array marks, if we know the address of marks[0] (1000), we can find the address of marks[1] by incrementing the address of marks[0] by 1(pointer increment). When we pass arrays by value, the function has a physically separate local copy. The function can access and even modify the array elements without effecting in the calling function. Example 7.1.6 passes the array marks to function selectionsort and sorting is done in the function. Selection sort is another method to sort data items

Example7.1.6: This example illustrates the concept of passing an array to a function. [059/c/5-b]

```
#include<stdio.h>
void selectionsort(int[],int);
void display_array(int[],int);
void main()
{
        int marks[5],temp,i,j;
        for(i=0;i<5;i++)
        {
                printf("Enter element %d  ",i);
                scanf("%d",&marks[i]);
        }
        selectionsort(marks,5);
        display_array(marks,5);
}
void selectionsort(int marks[],int n)
```

```
{
        int i,j,temp;
        for(i=0;i<n-1;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(marks[i]>marks[j])
                        {
                                temp=marks[i];
                                marks[i]=marks[j];
                                marks[j]=temp;
                        }
                }
        }
}
void display_array(int m[],int n)
{
        int i;
        printf("Sorted array is:");
        for(i=0;i<n;i++)
        {
                printf("%d ",m[i]);
        }
}
```

In example 7.1.6, main function passes an array named **marks** and its **size** to a function named **selectionsort**. The function sorts the array elements in ascending order. Similarly, main again calls another function named **display_array** to displays the sorted list. This sorting technique is selection sort. This can be done using bubble sort as in example 7.1.4.

7.1.2 Two-dimensional array

A two-dimensional array is a collection of data items, all of the same type, structured in two dimensions (referred to as rows and columns). Individual items are accessed by a pair of indices representing the position of each element.

Case Study : Let us consider a number of exams for a class of students. The score for each exam is to be weighted differently to compute the final score and grade. For example, the first exam may contribute 30% of the final score, the second may contribute 30%, and the third contribute 40%. We must compute a weighted average of the scores for each student. The sum of the weights for all the exams must add up to 1, i.e. 100%. Then we need to read the exam scores from a user or from file for several exams for a class of students. Read the percent weight for each of the exams. Compute the weighted average score for each student. Also, compute the averages of the scores for each exam and for the weighted average scores. We can think of the exam scores and the weighted average score for a single student as a data record and represent it as a row of information. The data records for a number of students, then, is a table of such rows. Here is our conceptual view of this collection of data in table 7.1.1

Table 7.1.1

Name	Exam1	Exam2	Exam3	Weighted avg
Student1	50	70	75	??
Student2	90	95	96	??
Student3	89	87	92	??
....	...	...	...	
....	....	...	...	
Student n	90	90	91	??

Let us assume that all scores will be stored as integers; even the weighted averages, which will be computed as float, will be rounded off and stored as integers. To store this information in a data structure, we can store each student's data record, a row containing three exam scores and the weighted average score, in a one-dimensional array of integers. The entire table, then, is an array of these one-dimensional arrays i.e., a two dimensional array. With this data structure, we can access a record for an individual student by accessing the corresponding row. We can also access the score for one of the exams or for the weighted average for all students by accessing each column. The only restriction to using this data structure is that all items in an array must be of the same data type. If the student id is an integer, we can even include a column for the id numbers.

Suppose, we need to represent id numbers, scores in 3 exams and weighted average of scores for 10 students; we need an array of ten data records, one for each student. Each data record must be an array of five elements, one for each exam score, one for the weighted average score, and one for the student id number. Then, we need an array, scores[10] that has ten elements; each element of this array is, itself, an array of 5 integer elements. Here is the declaration of an array of integer arrays:

scores[10][5];

The first range says the array has ten elements: **scores[0], scores[1],.......... scores[9]**. The second range says that each of these ten arrays is an array of five elements. For example, **scores[0]** has five elements: **scores[0][0], scores[0][1], scores[0][2], scores[0][3], scores[0][4]** and **scores[0][5]**. Similarly, any other element may be referenced

by specifying two appropriate indices i.e., **scores[i][j]**. The first array index references the one dimensional array, **scores[i]** ; the second array index references the element in the one dimensional array. A two dimensional array lend itself to a visual display in rows and columns. The first index represents a row and the second index represents a column. A visual display of the array, **scores[10][5]**, is shown in following figure 7.2.2. There are ten rows (0-9), and five columns (0-4). An element is accessed by row and column index. For example, **scores[2][3]** references an integer element at row index 2 and column index 3.

	column0	column1	column2	column3	column4
row0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]	scores[0][4]
row1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]	scores[1][4]
row2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]	scores[2][4]
row3	scores[3][0]	scores[3][1]	scores[3][2]	scores[3][3]	scores[3][4]
row4	scores[4][0]	scores[4][1]	scores[4][2]	scores[4][3]	scores[4][4]
row5	scores[5][0]	scores[5][1]	scores[5][2]	scores[5][3]	scores[5][4]
row6	scores[6][0]	scores[6][1]	scores[6][2]	scores[6][3]	scores[6][4]
row7	scores[7][0]	scores[7][1]	scores[7][2]	scores[7][3]	scores[7][4]
row8	scores[8][0]	scores[8][1]	scores[8][2]	scores[8][3]	scores[8][4]
row9	scores[9][0]	scores[9][1]	scores[9][2]	scores[9][3]	scores[9][4]

Fig. 7.2.2 Rows and columns in a two dimensional array.

Memory allocation process for two dimensional array is shown in figure 7.1.3. Here, for example,size of **score** array is taken 3X3.

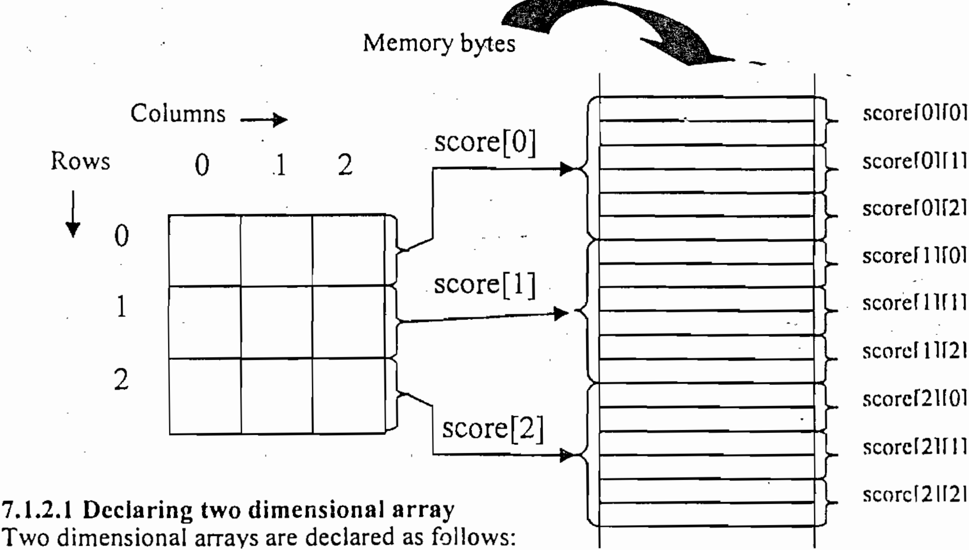


Fig 7.1.3 Illustration of memory allocation for each element of two-dimensional array

7.1.2.1 Declaring two dimensional array

Two dimensional arrays are declared as follows:

data_type array_name [row_size][column_size];

2D Arrays can be declared as local variables or global variables.

Global declaration

```
Char string[50][30]; /* Global declaration only */
void main()
{
 .
}
```

Local declaration

```
void main()
{
        float marks[20][10]; /* Local declaration only */
 .
}
```

7.1.2.2. Initializing two-dimensional arrays

we can initialize the elements of 2D array in the same way as the ordinary variables when they are declared.

Examples:

Global initialization

```
int marks[3][4] = {{2,3,5,4},{2,5,3,4},{4,3,3,2}}; /* Global initialization */
int marks[][4] = {{2,3,5,4},{2,5,3,4},{4,3,3,2}}; /* Global initialization */
void main()
{
 .
 .
}
```

Local initialization

```
vod main()
{
        int marks[3][4] = {   {2,3,5,4},
                              {2,5,3,4},
                              {4,3,3,2)
                                        }; /* Local initialization */

        int age[3][4]={{20,30,25,26},{23,45,34,46},{21,22,32,24}}; /* Local initialization */
    .
    .
    }
```

7.1.2.3 Accessing two-dimensional array elements

To access two-dimensional array we need to use the concept of rows and columns since array elements are arranged in a tabular form. Therefore, to access any element we need to use two loops: one for rows and one for columns. It will be more clear in the following examples.

Example 7.1.7: This program illustrates how to input and output elements of a two dimensional array.

```
#include<stdio.h>
void main()
{       int i,j,num[3][3];
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        printf("Enter num[%d][%d]:",i,j);
                        scanf("%d",&num[i][j]);
                        /* or scanf("%d",(*(num+i)+j)); */
                        /* (see chapter 7.3) */
                }
        }
        for(i=0;i<3;i++)
        {       for(j=0;j<3;j++)
                {       printf("%d ",num[i][j]);
                        /* or printf("%d",*(*(num+i)+j)); */
                }
                printf("\n");
        }
}
```

Output

```
Enter num[0][0]:1
Enter num[0][1]:2
Enter num[0][2]:3
Enter num[1][0]:4
Enter num[1][1]:5
Enter num[1][2]:6
Enter num[2][0]:7
Enter num[2][1]:8
Enter num[2][2]:9
1 2 3
4 5 6
7 8 9
```

Example 7.1.8: This program reads two two-dimensional arrays, adds the corresponding elements and displays the result on the screen.

```
#include<stdio.h>
void main()
{       int i,j,num1[10][10],r1,c1,num2[10][10],r2,c2,num3[10][10];
        printf("Enter the maximum size of row and column of array num1:");
        scanf("%d%d",&r1,&c1);
        printf("Enter the maximum size of row and column of array num2:");
        scanf("%d%d",&r2,&c2);
        if(r1==r2&&c1==c2)
        {
                for(i=0;i<r1;i++)
                {
                        for(j=0;j<c1;j++)
                        {
                                printf("Enter num1[%d][%d]:",i,j);
                                scanf("%d",&num1[i][j]);
                        }
                }
                for(i=0;i<r2;i++)
```

```
		{
			for(j=0;j<c2;j++)
			{
				printf("Enter num2[%d][%d]:",i,j);
				scanf("%d",&num2[i][j]);
			}
		}
		for(i=0;i<r1;i++)
		{
			for(j=0;j<c1;j++)
			{
				num3[i][j]=num1[i][j]+num2[i][j];
				printf("\t%d",num3[i][j]);
			}
			printf("\n");
		}
	}
	else
	{
		printf("Array size mismatch.");
	}
}
```

Output

```
Enter the maximum size of row and column of array num1:2 3
Enter the maximum size of row and column of array num2:2 3
Enter num1[0][0]:1
Enter num1[0][1]:2
Enter num1[0][2]:3
Enter num1[1][0]:4
Enter num1[1][1]:5
Enter num1[1][2]:6
Enter num2[0][0]:7
Enter num2[0][1]:8
Enter num2[0][2]:9
Enter num2[1][0]:0
Enter num2[1][1]:11
Enter num2[1][2]:12
    8    10    12
    4    16    18
```

7.1.2.4 Passing two-dimensional arrays to functions

Just as in one-dimensional array, when a two-dimensional (or higher dimensional) array is passed as a parameter, the base address of the actual array is sent to the function (passed by reference). Any change made to the elements of an array element inside a function will carry over to the original location of the array that is passed to the function. The size of all dimensions except the first must be included in the function heading and prototype. The sizes of those dimensions for the formal parameter must be exactly the same as in the actual array. The function header and prototype specify the number of columns as a constant. For example, the following function declarations are valid.

```
void function1( int x[8][5], int cs[] );
int function2( int x[][5], float m);
int function3( int[][5], float);
```

Example7.1.9: This program reads two two-dimensional arrays, passes them to a function to multiply, multiplies them and displays the result on the screen from the main function.

```
#include<stdio.h>
void mul2darray(int[][10], int[][10], int [][10], int, int, int);
void main()
{	int i,j,num1[10][10],r1,c1,k, num2[10][10],r2,c2,num3[10][10];
	printf("Enter the maximum size of row and column of array num1:");
	scanf("%d%d",&r1,&c1);
	printf("Enter the maximum size of row and column of array num2:");
	scanf("%d%d",&r2,&c2);
	if(c1==r2)
	{
		for(i=0;i<r1;i++)
		{
			for(j=0;j<c1;j++)
			{
				printf("Enter num1[%d][%d]:",i,j);
				scanf("%d",&num1[i][j]);
			}
		}
		for(i=0;i<r2;i++)
		{
			for(j=0;j<c2;j++)
			{
```

```
                        printf("Enter num2[%d][%d]:",i,j);
                        scanf("%d",&num2[i][j]);
                    }
                }
                mul2darray(num1,num2,num3,r1,c1,c2);
                for(i=0;i<r1;i++)
                {
                    for(j=0;j<c2;j++)
                    {
                        printf("%d ",num3[i][j]);
                    }
                    printf("\n");
                }
        }
        else
        {
                printf("Array size mismatch.");
        }
}
void mul2darray(int num1[][10],int num2[][10],int num3[][10],int r1,int c1,int c2)
{
        int i,j,k;
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c2;j++)
                {
                        num3[i][j]=0;
                        /*  *(*(num3+i)+j)=0; */
                        for(k=0;k<c1;k++)
                        {
                            num3[i][j]=num3[i][j]+num1[i][k]*num2[k][j];
                            /* *(*(num3+i)+j)=*(*(num3+i)+j)+(*(*(num1+i)+k))*(*(*(num2+k)+j)); */
                        }
                        /* printf("\t%d",num3[i][j]); */
                }
        }
}
```

Output

```
Enter the maximum size of row and column of array num1:3 3
Enter the maximum size of row and column of array num2:3 3
Enter num1[0][0]:0
Enter num1[0][1]:2
Enter num1[0][2]:3
Enter num1[1][0]:5
Enter num1[1][1]:0
Enter num1[1][2]:7
Enter num1[2][0]:1
Enter num1[2][1]:0
Enter num1[2][2]:1
Enter num2[0][0]:2
Enter num2[0][1]:0
Enter num2[0][2]:5
Enter num2[1][0]:3
Enter num2[1][1]:0
Enter num2[1][2]:0
Enter num2[2][0]:5
Enter num2[2][1]:3
Enter num2[2][2]:1
21  9  3
45  21  32
7  3  6
```

7.1.3 Multidimensional arrays

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is

data type array_name [s1][s2][s3][s4]......[sn];

Where si is the size of the ith dimension. Some examples are:

int num[3][10][4][5];

where num is a four dimensional array declared to contain 600 integer type elements. Other procedures are similar to the one and two-dimensional arrays.

7.1.4 Some important examples

Example 7.1.10: Write a program that adds the individual rows of a two dimensional array of m by n and store the sums of rows into a single dimensional array using functions. Write a function that takes a two-dimensional array and one-dimensional array and process the result and store in one-dimensional array. [2062/ b]

```
#include<stdio.h>
void onedandtwodarray(int [][3],int[],int,int);
void main()
{
        int twodarray[2][3],onedarray[2],r,c,i,j;
        for(i=0;i<2;i++)
        {       for(j=0;j<3;j++)
                {
                        printf("enter twodarray[%d][%d]",i,j);
                        scanf("%d",&twodarray[i][j]);
                }
        }
        onedandtwodarray(twodarray,onedarray,2,3);
        for(i=0;i<2;i++)
        {
                printf("onedarray[%d]=%d\n",i,onedarray[i]);
        }
}
void onedandtwodarray(int a2d[][3],int a1d[],int p,int q)
{
        int i,j;
        for(i=0;i<p;i++)
        {     a1d[i]=0;
                for(j=0;j<q;j++)
                {

                        a1d[i]=a1d[i]+a2d[i][j];
                }
        }
}
```

Example 7.1.11: Write a program that reads two matrices of order mxn and pxq using a function readMatrix(). The program should contain a function processMartrix() that takes the matrices and multiplies them. The result of multiplication must be displayed using a function showMatrix(). [2060/p/8, 058/c/4-b]

```
#include<stdio.h>
int row,col;
void readMatrix (int a[][10], int b[][10], int m, int n, int n1)
{
        for(row=0;row<m;row++)
                for(col=0;col<n;col++)
                {
                        printf("\nEnter a[%d][%d] element of matrix a",row,col);
                        scanf("%d",&a[row][col]);
                }
        for(row=0;row<n;row++)
                for(col=0;col<n1;col++)
                {
                        printf("\nEnter b[%d][%d] element of matrix b",row,col);
                        scanf("%d",&b[row][col]);
                }
}
void processMatrix (int a[][10], int b[][10], int c[][10], int m, int n, int n1)
{
        int k;
        for(row=0;row<m;row++)
        {
                for(col=0;col<n1;col++)
                {
                        c[row][col]=0;
                        for(k=0;k<n;k++)
                        {
                                c[row][col]+=a[row][k]*b[k][col];
                        }
                }
        }
}
void showMatrix (int c[][10],int m,int n1)
{
        for(row=0;row<m;row++)
        {
                for(col=0;col<n1;col++)
                printf("\t%d",c[row][col]);
```

```
                printf("\n");
        }
}
void main()
{
        int m,n,p,q;
        int a[10][10],b[10][10],c[10][10];
        printf("Enter number of rows and columns for first matrix:");
        scanf("%d%d",&m,&n);
        printf("Enter number of rows and columns for second matrix:");
        scanf("%d%d",&p,&q);
        if(n==p)
        {
                readMatrix (a,b,m,n,q);
                processMatrix (a,b,c,m,n,q);
                showMatrix (c,m,q);
        }
        else
                printf("Matrix multiplication condition is not satisfied!!!..\n Try Again.");
}
```

Exercise 7.1

1. What is an array? How memory is allocated for a one-dimensional array? Explain with a suitable diagram.
2. Explain briefly about local and global declaration and initialization of one-dimensional and two-dimensional arrays with relevant examples.
3. How can we access the elements of one-dimensional and two-dimensional arrays?
4. Differentiate between sorting and searching of array elements.
5. How can we pass the one-dimensional and two-dimensional arrays to function? Illustrate with a suitable examples
6. Write a program to count and find the sum of all the numbers in the array which are exactly divisible by 7 but not by 5. [062/b/2-b]

   Hint: `if(array[i]%7==0 && array[i]%5!=0)`
   `sum=sum+array[i];`
7. Write a computer program to raise the power of each elements of given matrix of order MxN
8. by 2. Display resultant matrix. [057/c/5-b]
9. Write a function that takes an one dimensional array of 'n' elements and sorts them in ascending order. Test the function in your program [060/P/4-b]
10. Write a program to find the sum of squares of elements on the diagonal of a square matrix.
11. Write a program to sort each row of a two dimensional array of size p by q.
12. Write a program to transpose a two dimensional matrix of size m by n.
13. Write a program that finds the sum and difference of the largest and smallest element of a two dimensional array of size m by n.
14. The annual examination results of 100 students are tabulated as follows:

Roll No.	Subject 1	Subject 2	Subject 3	Subject 4

Write a program to read the data and determine the following:

a. Total marks obtained by each student.
b. The highest marks in each subject and the Roll No. of the student who secured it.
c. The student who obtained the highest total marks.

15. Write a program to find the average of each row and column elements of a two dimensional array of size m X n.

7.1.4 Some important examples

Example 7.1.10: Write a program that adds the individual rows of a two dimensional array of m by n and store the sums of rows into a single dimensional array using functions. Write a function that takes a two-dimensional array and one-dimensional array and process the result and store in one-dimensional array. [2062/ b]

```
#include<stdio.h>
void onedandtwodarray(int [][3],int[],int,int);
void main()
{
	int twodarray[2][3],onedarray[2],r,c,i,j;
	for(i=0;i<2;i++)
	{	for(j=0;j<3;j++)
		{
			printf("enter twodarray[%d][%d]",i,j);
			scanf("%d",&twodarray[i][j]);
		}
	}
	onedandtwodarray(twodarray,onedarray,2,3);
	for(i=0;i<2;i++)
	{
		printf("onedarray[%d]=%d\n",i,onedarray[i]);
	}
}
void onedandtwodarray(int a2d[][3],int a1d[],int p,int q)
{
	int i,j;
	for(i=0;i<p;i++)
	{	a1d[i]=0;
		for(j=0;j<q;j++)
		{

			a1d[i]=a1d[i]+a2d[i][j];
		}
	}
}
```

Example 7.1.11: Write a program that reads two matrices of order mxn and pxq using a function readMatrix(). The program should contain a function processMartrix() that takes the matrices and multiplies them. The result of multiplication must be displayed using a function showMatrix(). [2060/p/8, 058/c/4-b]

```
#include<stdio.h>
int row,col;
void readMatrix (int a[][10], int b[][10], int m, int n, int n1)
{
	for(row=0;row<m;row++)
		for(col=0;col<n;col++)
		{
			printf("\nEnter a[%d][%d] element of matrix a",row,col);
			scanf("%d",&a[row][col]);
		}
	for(row=0;row<n;row++)
		for(col=0;col<n1;col++)
		{
			printf("\nEnter b[%d][%d] element of matrix b",row,col);
			scanf("%d",&b[row][col]);
		}
}
void processMatrix (int a[][10], int b[][10], int c[][10], int m, int n, int n1)
{
	int k;
	for(row=0;row<m;row++)
	{
		for(col=0;col<n1;col++)
		{
			c[row][col]=0;
			for(k=0;k<n;k++)
			{
				c[row][col]+=a[row][k]*b[k][col];
			}
		}
	}
}
void showMatrix (int c[][10],int m,int n1)
{
	for(row=0;row<m;row++)
	{
		for(col=0;col<n1;col++)
		printf("\t%d",c[row][col]);
```

Chapter 7.2
Pointers

7.2.1 Introduction

Till this point, we have have used variable name to access the value of a variable. So, we have not cared about the physical location of our data within memory. The memory of our computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one. This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1000 we know that it is going to be right between cells 999 and 1001. The concept of memory organization is illustrated in the figure 7.2.1.

Memory cells	Address
	0
	1
	2
	3
	.
	.
	.
	.
	.
	.
	.
	.
	.
	.
	.
	.
	.
	.
	65535

Fig.7.2.1 Organization of 64KB memory

The concept of variable representation

The figure 7.2.2 shows that how a variable is represented in memory. Each variable have two values: one is its address in memory (lvalue) and another is its scontent (rvalue).

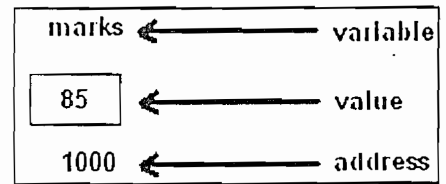


Fig. 7.2.2 representation of a variable

Pointer variables

A pointer is a variable, which contains the address of another variable in memory. We can have a pointer to any variable type. Pointers are said to "point to" the variable whose reference they store. Pointers are a very powerful feature of the C language that has many uses in advanced programming. The figure7.2.3 shows the concept of a pointer variable Where **marks_pointer** is a pointer variable which stores the address of marks i,e, it is pointing to the variable marks.

Variable	value	Address
marks	85	1000
marks_pointer	1000	1030

Fig. 7.2.3 representation of a pointer variable

7.2.4 Reference operator (&)

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator and which can be literally translated as "address of". For example, in figure7.2.3, the expression **marks_pointer=&marks** assigns the address of marks to **marks_pointer**.

7.2.5 Dereference operator (*)

Using a pointer we can directly access the value stored in the variable, which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by". Thus, & and * have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *. For example, the value of marks can be obtained as

***marks_pointer= *(&marks)=marks;**

7.2.6 Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point. It is not the same thing to point to a `char` than to point to an int or a float.

The declaration of pointers follows this format:

data_type * pointer_variable_name;

where **data_type** is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself but the type of the data the pointer points to. For example, in the above case marks pointer must be declared as

int *marks_pointer;

Similarly, other types can be declared as:

int * number;
char * character;
float * great_number

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space and they are not of the same type: the first one points to an int, the second one to a char and the last one to a float. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: int*, char* and float* respectively, depending on the type they point to.

7.2.7 Pointer initialization

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as

int mark;
int *marks_ptr;
marks_ptr=&marks;

Here, **marks_ptr** is a pointer variable that points the ordinary variable marks. Before using pointer variables, they should be initialized. When a pointer initialization takes place, we are always assigning the reference value to where the pointer points, never the value being pointed.

7.2.8 Pointer expressions

Like ordinary variables, pointer variables can be used in expressions. For examples, if x1 and x2 are properly declared and initialized pointers, they can be used in the expressions in the following forms.

```
p=*x1 * *x2;
sum=sum+*x2;
*x1= *x1 + 40;
x=10* - *x1/ x2  same as(10 * (-(*x1)))/(*x2)
```

Example 7.2.1: The following example declares two normal variables and two pointer variables, which point to the normal variables.

```
#include<stdio.h>
void main()
{
        int *p,*q; /* Declaration of pointer variables */
        int a,b;  /* Declaration of ordinary variables */
        p=&a; /* Using referencing operator to initialize pointer variable p. */
        q=&b;
        printf("Address of a=%u\n",&a);
        printf("Address of b=%u\n",&b);
        printf("value of p=%u\n",p);
        printf("value of q=%u\n",q);
        printf("Enter value of a and b:");
        scanf("%d%d",&a,&b);
        /*Using dereferencing operator (*).*/
        printf("The value pointed by p is %d\n",*p);
        printf("The value pointed by q is %d\n",*q);
        printf("a+b=%d\n",a+b);
        printf("*p+*q=%d",*p+*q);
        /* *p+*q → pointer expression */
}
```

Output

```
Address of a=65524
Address of b=65522
value of p=65524
value of q=65522
Enter value of a and b:45 67
The value pointed by p is 45
The value pointed by q is 67
a+b=112
```

*p+*q=112

Example 7.2.2: An example to show the concept of pointer expressions.

```
#include<stdio.h>
void main()
{
        float *p,*q,a,b,mul,x,y;
        p=&a;
        q=&b;
        printf("Enter value of a and b:");
        scanf("%f%f",&a,&b);
        x=*p * *q/3-10;
        mul=*p * *q;
        y=10+*q;
        *p=*p+*q;
        printf("x=%f\n",x);
        printf("product=%f\n",mul);
        printf("y=%f\n",y);
        printf("*p=%f\n",*p);
}
```

Output

```
Enter value of a and b: 40 30
x=390.000000
product=1200.000000
y=40.000000
*p=70.000000
```

7.2.9 Pointer arithmetic

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. All pointer arithmetic operations that involve two pointers assume both pointers are of the same type. For example, let us use the following segment of code:

```
int u,v, *p,*q;
```

Table 7.2.1 Scale factor of different data type

Data type	Scale factor
char	1
int	2
float	4
double	8
long int	4
long double	10

The permissible arithmetic operations on pointer variables like p and q are summarized below:

- A pointer variable can be assigned the address of an ordinary variable(e.g., p=&u).
- A pointer variable can be assigned the address of another variable(e.g., p= q) provided both pointers point to objects of the same data type.
- A pointer variable can be assigned a null(zero) value (e.g. p= NULL, where NULL is a symbolic constant that represents the values 0).
- An integer quantity can be added to or subtracted from a pointer variable (e.g., p+3, ++v, p-3, --v etc.)
- One pointer variable can be subtracted from another provided both pointers point to elements of the same array.
- Two pointers variables can be compared provided both pointers point to the elements of the same data type.

Other arithmetic operations on pointers are not allowed. For example,

- Pointer variables can not be multiplied or divided by a constant.
- Two pointers variables can not be added.
- An ordinary variable can not be assigned an arbitrary address.

7.2.10 Pointer increments and scale factor

As we have seen in the fourth point of pointer arithmetic, an integer quantity can be added to or subtracted from a pointer variable. Now we use the concept to increment or decrement a pointer. A pointer p1 can be incremented like

```
p1=p2+2;
p=p+1;
```

and so on.

However, an expression like p++ will cause the pointer p to point to the next value of its type. For example, if p is an integer pointer with an initial value 1000, then after the execution of statement p=p+1; the value of p will be 1002, but not 1001. That is, when we increment a pointer, its value is incremented by the size of the data type that it points to. The size of data type is called the scale factor. The scale factors of various data types are shown in the table 7.2.1.

7.2.11. Pointers and functions

We can pass the address of a variable as an argument to a function. When we pass address to a function, the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as call by reference. The function, which is called by reference, can change the value of the variable used in the call. While calling a function by reference, the points to be noted are:

- The function parameters must be declared as pointers.
- The dereferenced pointers are used in the function body.
- When the functions are called the addresses of the arguments are passed as actual arguments.

Example7.2.3: A program to find the larger of two numbers using the concept of function and pointer.

```
#include<stdio.h>
void larger(int*,int*,int*);
```

```
void main()
{
        int a,b,large;
        printf("Enter evalues of a and b:");
        scanf("%d%d",&a,&b);
        printf("&a=%u\n&b=%u\n&larger=%u\n",&a,&b,&large);
        /* This line is to print the address of a, b and large in main. */
        larger(&a,&b,&large);
        printf("larger=%d", large);
}
void larger(int*p,int*q,int *l)
{
        printf("p=%u\nq=%u\nl=%u\n",p,q,l);
        /* This line is to print the received */
        /* address in function */
        /* which is same as in the main. */
        if(*p>*q)
                *l=*p;
        else
                *l=*q;
}
```

Output

```
Enter values of a and b: 40  12
&a=65524
&b=65522
&larger=65520
p=65524
q=65522
l=65520
larger=40
```

Question : Why return statement is not required in example 7.2.3?

Example 7.2.4 : This example illustrates about function returning pointer as will as reversing the array elements.

```
#include<stdio.h>
int* reverse_array();
void main()
{
        int i,*p;
        p=reverse_array();
        printf("Reversed array is:");
        for(i=0;i<5;i++)
        {
                printf("%d\t",p[i]);
        }
}
int* reverse_array()
{
        int i,temp;
        static int num[5]={10,20,30,40,50};
        for(i=0;i<5/2;i++)
        {
                temp=num[i];
                num[i]=num[4-i];
                num[4-i]=temp;
        }
        return num;
}
```

Output

Reversed array is: 50 40 30 20 10

7.2.12. Arrays and pointers

Pointers and arrays are inseparably related, but they are not the synonyms. An array is a non empty set of sequentially indexed elements having the same type of data. Each element of an array has a unique identifying index number. In C, the declaration **float marks[6];** defines an array of size 6, each of which is an float object. These objects named marks[0], marks[1], ..., marks[6] appear contiguously in the storage area, that is, there is no padding between the array members. The base address is the location of the first element (index 0) of the array. The compiler defines the array name as a constant pointer to the first element. Suppose, the base address of array marks is 1000 and assuming each float memer requires four bytes then six elements will be stored as in figure 7.2.4.

Base address(&marks[0])

address	1000	1004	1008	1012	1016	1020
value	90	80	76	45	75	85
elements	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]

Fig.7.2.4 Illustration of memory allocation of an array

The array name marks is defined as a constant pointer pointing to the first element, marks[0] and therefore the values of marks is 1000. That is, marks=&marks[0]=1000, then, if we declare p as an float pointer, then we can make the pointer p to point to the array marks by the following assignment statement:

p=marks;

This is equivalent to **p=&marks ;**
Now, we can access every value of marks using p++ to move from one element to another. The relationship of p and marks is follows:

p=&marks[0](=*1000*)
p+1=&marks[1](=*1004*).......p+5=&marks[5](=*1020*).

We know that the address of the element is calculated using the index and the scale factor of the data type. For example,
address of marks[4]=base address+(4*scale factor of float)
= 1000+(4*4)=1016
When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that *(p+4) gives the value of marks[4]. We need not required to assign the base address of the array to another pointer variable. We can use the array name as a pointer. So that, instead of *(p+4), we can user *(marks+4). The programs using pointer notations are faster than array notation. In general, the above relation is shown in the table 7.2.2.

Table 7.2.2 equivalent expressions of arrays and pointers

To be accessed (↓)	Technique (→)	Array notation	Pointer notation
Address of ith element		&marks[i]	(marks+i)
Value of ith element		marks[i]	*(marks+i)

Similarly, pointers can also be used to manipulate two-dimensional arrays. Suppose, if we have a two dimensional array:
float marks[5][10];
As in the one-dimensional array, the equivalent pointer expressions for two-dimensional arrays are shown in table 7.2.3. (see chapter 7.3.6)

Table 7.2.3 equivalent expressions of arrays and pointers

To be accessed (↓)	Technique (→)	Array notation	Pointer notation
Address of element at i[th] row and j[th] column.		&marks[i][j]	(*(marks+i)+j)
Value of element at i[th] row and j[th] column		marks[i][j]	*(*(marks+i)+j)

WARNING: There is no bound checking of arrays and pointers so we can easily go beyond array memory and overwrite other things.

However pointers and arrays are different:

A pointer is a variable. We can do **p = marks, p++** etc.
An Array is not a variable. **marks = p and marks++** are illegal. In case of array, there occurs static memory allocation but in case of pointers, there occurs dynamic memory allocation. Difference in element access method.(see tables 7.2.2 and 7.2.3)As we see there is no difference for the coder, when accessing an element in array, using the [] or the * expression. The * expression could lead to a much more efficient implementation then the [] expression.

Example 7.2.5: This example uses pointer notation to access array elements.

```
#include<stdio.h>
void main()
{
        int age[10],i;
        printf("Enter elements:");
        for(i=0;i<10;i++)
        {
                scanf("%d",age+i);
        }
        for(i=0;i<10;i++)
        {
                printf("%d ", *(age+i) );
        }
}
```

> As in example 7.2.4, we can use pointers expressions in all the examples that have done using arrays in chapter 7.1.

7.2.13. Pointers to pointers

C allows the use of pointers that point to pointers. In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations.
Examples:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as in figure 7.2.5. The value of each variable is written inside each cell; under the cells are their respective addresses in memory. The new thing in this example is variable **c**, which can be used in three different levels of indirection; each one of them would correspond to a different value:

- c has type char** and a value 8092
- *c has type char* and a value 7230
- **c has type char and a value 'z'

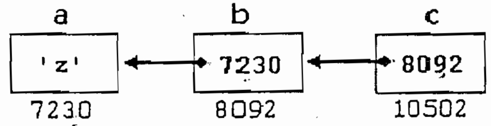


Fig. 7.2.5 illustration of pointers to pointers

7.2.14. Arrays of pointers

We can have arrays of pointers since pointers are variables.
e.g. **int *p[10];**
Which means that p is an array of 10 pointer which points to integer data type.

7.2.15. void pointer

The void type of pointer is a special type of pointer. In C, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties). This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to change the type of the void pointer to some other pointer type that points to a concrete data type before dereferencing it. This is done by performing type-casting. We can not apply pointer arithmetic to void pointers. (see chapter 7.3)

Example 7.2.6 This example illustrates the concept of void pointer.

```
#include<stdio.h>
void main()
{
        void *p,*q;   /* void pointer variables declaration */
        int a=33,b=44,c;    /* ordinary variables declaration and initialization */
        p=&a,q=&b;        /* pointers initialization */
        c=*(int*)p+*(int*)q; /* converting void poiners to integer pointer using type casting (int*) */
        printf("a+b=%d",c);
}
```

Output
a+b=77

7.2.16. Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address.

```
int * p;
p = 0;
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type.

7.2.17. Reasons for using pointers

- A pointer enables us to access a variable that is defined outside the function.
- Pointers are more efficient in handling the data tables.
- The use of pointer array to characters strings results in saving of data storage space in memory.
- It is the only way to express some computations.
- It produces compact, efficient and powerful code with high execution speed .
- Pointers are used in dynamic memory allocation. They are used to store the addresses created by malloc(), calloc(), realloc() during execution
- Pointers are used to return more than one values from a function when parameters are passed by reference. They are also used to pass arrays to functions.
- Pointers are used in the manipulation of objects, particularly during sorting.

Example 7.2.7: A program that takes three variables (a, b, c) and rotates the values stored such that values of a goes to b, b to c and c to a. Use pointer for rotation.

```
#include<stdio.h>
void main(void)
{
        int a,b,c,temp;
        int *d,*e,*f;
        d=&a,e=&b,f=&c;
        printf("Enter three numbers:");
        scanf("%d%d%d",&a, &b,&c);
        printf("\nBefore rotation %d %d %d",*d,*e,*f);
        temp=*f;
        *f=*e;
        *e=*d;
        *d=temp;
        printf("\nAfter rotation %d %d %d",*d,*e,*f);
}
```

7.2.18. Review

- A ordinary variable is declared by giving it a type and a name (e.g., int k;)
- A pointer variable is declared by giving it a type and a name (e.g. int *ptr) where the asterisk tells the compiler that the variable named ptr is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).
- The FIRST thing we must do to use a pointer is to initialize it
- Once a variable is declared, we can get its address by preceding its name with the unary & operator, as in &marks. We can "dereference" a pointer, i.e., refer to the value of that which it points to, by using the unary '*' operator as in *marks.
- The "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored at that address.

Exercise 7.2

1. What is a pointer? What are the main reasons of using pointers? [057/c/4-a]
2. Differentiate between ordinary variables and pointers variables.
3. Differentiate between references and dereference operators.
4. What is pointer initialization? Why is it required? Illustrate with an example.
5. How can we write pointer expressions? Explain with an example.
6. State the main difference between *p and p in expression **int *p.**
7. Explain along with examples about the pointer arithmetic. [061/p/6-a, 062/p/11]
8. What is scale factor? What is its role in pointer increment?
9. What is the role of pointers when passing arrays to functions?
10. Differentiate between null and void pointers.
11. How do you relate array and pointer, illustrate with an example.[2057/c/4-a,058/c/,059/c/,062/b]
12. Write down the similarities and differences between arrays and pointers. [062/b]
13. Compare array and pointer with example. [2063/b/5]
14. What do you understand by pointer? Explain its use in function and dynamic memory allocation. [2063/p/4]

Chapter 7.3
Dynamic Memory Allocation

7.3.1 Some introductory terms

Compile time: It is the time when a compiler compiles code written in a programming language into an executable form.

Runtime: It describes the operation of a computer program, the duration of its execution, from beginning to termination.

Computer storage or computer memory: It refers to the computer components, devices and recording media that retain binary information for some interval of time. Usually, memory refers to forms of storage which are fast but lose their contents in a case of power loss and storage refers to forms of storage which are slower but suitable for long-term retention.

Automatic variables: These are variables local to a block. They are automatically allocated on the stack when that block of code is entered. When the block exits, the variables are automatically deallocated.

Stack: It is a data structure that works on the principle of Last In First Out (LIFO). This means that the last item put on the stack is the first item that can be taken out, like a physical stack of plates

Static memory allocation: It refers to the process of allocating memory at compile-time before the associated program is executed, unlike dynamic memory allocation where memory is allocated as required at run-time

Garbage collection (GC): It is a form of automatic memory management. The garbage collector or collector attempts to reclaim the memory used by objects that will never be accessed again by the application.

Memory leak: It is often thought of as a failure to release unused memory by a computer program. Strictly speaking, it is just unnecessary memory consumption. A memory leak occurs when the program loses the ability to free the memory. A memory leak diminishes the performance of the computer, as it becomes unable to use all its available memory.

7.3.2. Dynamic memory allocation

So far we have considered static memory allocation through the declaration of variables. Now, we consider dynamic memory allocation, which allows a program to obtain more memory space, while running or to release space no longer required. This is important for array based programs where size of array is not known at the compile time. C language requires the numbers of elements in an array to be specified compile time. But we may not be able to do so always. If our initial judgment of size is wrong, it may cause failure of the program or wastage of memory spaces. When writing a program that is either time or space constrained it may be useful to create variables only when needed. So, dynamic memory allocation is the process of allocating memory storage for use in a computer program during the runtime of that program. It is a way of distributing ownership of limited memory resources among many pieces of data and code. A dynamically allocated object remains allocated until it is deallocated explicitly, either by the programmer or by a garbage collector. That is notably different from static memory allocation. We say that such an object has dynamic life time. Failure to deallocate the memory creates "memory leak".

7.3.3. Memory allocation process

Figure 7.3.1 shows the conceptual view of storage of a C program in memory. The program istructions, global and static variables are stored in a region known as a permanent storage area and local variables are stored in another area called stack. The memory space that is located between these two is available for dynamic allocation during the execution of the program. This free memory is called heap. The size of heap keeps changing when program is executed due to the creation and death of variables that are local to functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situation, the memory allocation functions described below return a NULL pointer.

Memory	Region
Local variables	stack
Free memory	Heap
Global variables	Permanent storage area
C program instructions	Permanent storage area

Fig.7.3.1 Storage of a C program

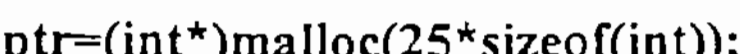
7.3.4. Memory management functions

Although C does not inherently have this facility, there are four library functions known as "memory management functions" that can be used for allocating and deallocating (freeing) memory during program execution. The functions are:

a. malloc
b. calloc
c. free
d. realloc

7.3.4.1 malloc

A block of contiguous memory can be allocated using the function malloc. The malloc function reserves a block of memory of specified size and return a pointer of type void. This means that it can be casted to any type of pointer. Its general form is:
ptr=(cast-type*)malloc(byte-size);



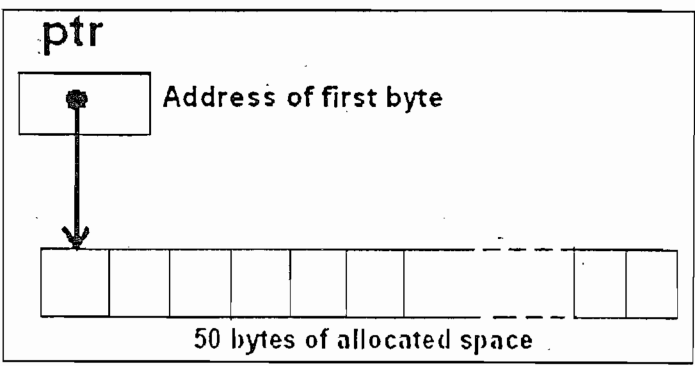


Fig.7.3.2 Allocated space and a pointer pointing the space

Where ptr is a pointer of cast type. The malloc returns a pointer (of cast type) to an area of memory with size of byte-size. If the space in the heap is not sufficient to satisfy the request, the allocation can fail. If it fails, it returns a NULL pointer.
Example:

ptr=(int*)malloc(25*sizeof(int));

On successful execution of the statement, a memory space equivalent to 50 bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **ptr** of type int. Which is shown in figure7.3.2. Note that, the memory allocated dynamically has no name and therefore its contents can be accessed through a pointer.

7.3.4.2 calloc

This function is normally used for allocating memory space at run time for storing derived data typed such as array and structures. While malloc allocates single block of memory, but calloc allocates multiple blocks of memory each of the same size and sets all bytes to zero. Like malloc, it also returns void pointer on successful allocation of requested memory area. The void pointer must be casted to point to a particular type. Its general form is:

ptr=(cast-type*)calloc(n,sizeof (data type of the elements));
For example: **ptr=(float*)calloc(25,sizeof(float));**

This statement allocates contiguous space for 25 blocks each of size four bytes, initializes all bytes to zero and returns the pointer of the first byte to **ptr**. If there is not enough space, calloc returns NULL pointer. Memory allocation process of **calloc** function is visualized in figure 7.3.3.

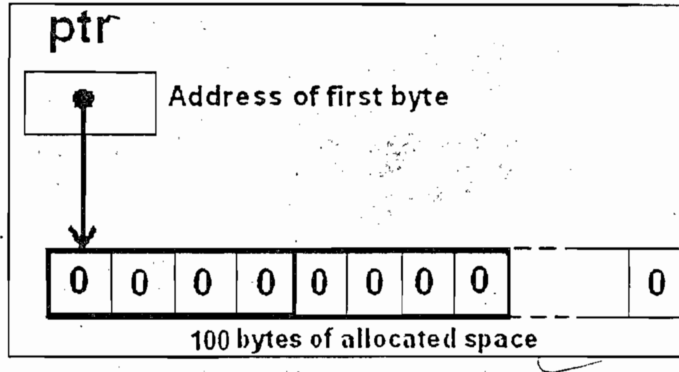


Fig.7.3.3 Allocated space and a pointer pointing the sp

7.3.4.3 free

The **free** function causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. To use free function we do:

free(ptr);

wehere ptr is a pointer to a memory block which has already been created by malloc or calloc. Use of invalid pointer in the call may create problems.

7.3.4.4 realloc

What will we do?
I - If previously allocated memory is not sufficient and we need additional space for more elements,
Ii - if previously allocated space is much more than required and we want to reduce it.
In such situations, we can change the memory size already allocated with the help of the function realloc. This process is called the reallocation of memory. For example, if the original allocation is done by the statement

ptr=malloc(size);

then reallocation of space may be done by the statement

ptr=realloc(ptr,newsize);

This statement allocated a new memory space of size **newsize** to the pointer variable ptr and returns a pointer to the first byte of the new memory block. The newsize may be larger or smaller then the **size**. The new memory block may or may not begin at the same place as the old one. The function guarantees that the old data will remain intact.

Table 7.3.1 Summary of memory management functions

Function	Task
malloc	Allocated requested size of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	frees the previously allocated space.
realloc	Modifies the size of previously allocated space.

7.3.5. Some important examples

The following examples use the above-described memory management functions to allocate memory dynamically in one -dimensional arrays. All the other operation can be done as in one-dimensional array except array declaration.

Example 7.3.1: This program asks the required size of array to the user and displays the addresses of allocated blocks.

```
#include<stdio.h>
#include<alloc.h>
void main(void)
{
        int n,i;
        float *address;
        printf("Enter number of elements:");
        scanf("%d",&n);
        address=(float*)calloc(n,sizeof(float));
        if(address==NULL)
        {       printf("Memory can not allocated.");
                exit();
        }
        for(i=0;i<n;i++)
        {
                printf("\nAddress of %d block %d ",i,(address+i));
        }
        free(address);
}
```

Output
Enter number of elements:6
Address of 0 block 2558
Address of 1 block 2562
Address of 2 block 2566
Address of 3 block 2570
Address of 4 block 2574
Address of 5 block 2578

Example 7.3.2: This program reads array elements and displays them with their corresponding storage locations. Sometimes memory can not be allocated due to lack of requested amount of free memory. We must handle this situation. So, we should check whether the required memory is allocated or not in every program. If the memory can not be allocated, the memory allocation function returns NULL pointer to the pointer variable. Study the following example and understand how the above discussed concept is handled.

```
#include<stdio.h>
void main(void)
{
        int n,i;
        float *address;
        printf("Enter number of elements:");
        scanf("%d",&n);
        if((address=(float*)malloc(n*sizeof(float)))==NULL)
        {
                printf("Memory can not be allocated.");
                exit();
        }
        printf("Enter array elements:");
        for(i=0;i<n;i++)
        {       scanf("%f",(address+i));
        }
        for(i=0;i<n;i++)
        {       printf("\nElement at %u is %f ",(address+i),*(address+i));
        }
        free(address);
}
```

Output
Enter number of elements:6
Enter array elements:1 3 7 33 45 78 90
Element at 2574 is 1.000000
Element at 2578 is 3.000000
Element at 2582 is 7.000000
Element at 2586 is 33.000000
Element at 2590 is 45.000000
Element at 2594 is 78.000000

Example 7.3.3: This program sorts a dynamic array using concept of both function and pointer.

```
#include<stdio.h>
void sort(float*,int);
void main(void)
{
        int n,i;
        float *marks;
        printf("Enter number of elements:");
        scanf("%d",&n);
        if((marks=(float*)calloc(n,sizeof(float)))==NULL)
        exit();
        printf("Enter array elements:");
```

```
        for(i=0;i<n;i++)
        {       scanf("%f",(marks+i));
        }
        sort(marks,n);
        for(i=0;i<n;i++)
        {
                printf("\nElement at %u is %f ",(marks+i),*(marks+i));
        }
        free(marks);
}
void sort(float *m,int n)
{       int pass,j,i,temp;
        for(pass=0;pass<n-1;pass++)
        {
                for(j=0;j<n-pass-1;j++)
                {
                        if(*(m+j)<*(m+j+1))
                        {
                                temp=*(m+j+1);
                                *(m+j+1)=*(m+j);
                                *(m+j)=temp;
                        }
                }
        }
}
```

Output

```
Enter number of elements:5
Enter array elements:123   456   111   111   23
Element at 2574 is 456.000000
Element at 2578 is 123.000000
Element at 2582 is 111.000000
Element at 2586 is 111.000000
Element at 2590 is 23.000000
```

Example 7.3.4: Write a program to read an array of n elements. The program has to allocate memory dynamically and pass the array to a function, which has to find the smallest, the largest numbers and average of all elements and pass to the calling function. Use pass by reference if necessary. [2062/5/b, similar to (2058/p/6-b,061/b/6-b, 061/p/6/b)]

```
#include<stdio.h>
void findminmax(int *,int *,int *,int, float*);
void main()
{
        int largest,smallest,n,*p,i;
        float average;
        printf("Enter array size:");
        scanf("%d",&n);
        p=(int*)calloc(n,sizeof(int));
        if(p==NULL)
        exit();
        for(i=0;i<n;i++)
        {
                printf("Enter array element [%d]:",i+1);
                scanf("%d",(p+i));
        }
        findminmax(p,&largest,&smallest,n,&average);
        printf("Largest=%d\nSmallest=%d\nAverage=%f",largest,smallest,average);
        free(p);
}

void findminmax(int *array,int *l,int *s, int n, float *avg )
{
        int i, sum=array[0]; /*because in the following loop i is started form 1.*/
        *l=*(array+0);
        *s=*(array+0);
        for(i=1;i<n;i++)
        {
                if(*l<*(array+i))
                *l=*(array+i);
                if(*s>*(array+i))
                *s=*(array+i);
                sum=sum+*(array+i);
        }
        *avg=(float)sum/n;
}
```

Output

```
Enter array size:5
Enter array element [1]:12345
Enter array element [2]:23
Enter array element [3]:354
Enter array element [4]:56
Enter array element [5]:87
Largest=12345
Smallest=23
Average=7573.000000
```

Example 7.3.5 Write a program that reads two one dimensional arrays of order m and n respectively and merge the contain of first and second array to third array of order (m+n) and display it. [058/c/8]

(While allocating memory dynamically, it is required to check whether the requested amount is allocated or not. How to do so and handle the condition ?)

```
#include<stdio.h>
#include<alloc.h>
void main(void)
{           int *p,*q,*r,m,n,i,j;
            clrscr();
            printf("\nEnter the size(m) of array P:");
            scanf("%d",&m);
            printf("Enter the size(n) of array q:");
            scanf("%d",&n);
            p=(int *)calloc(m,sizeof(int));
            q=(int *)calloc(n,sizeof(int));
            r=(int *)calloc(m+n,sizeof(int));
            for(i=0;i<m;i++)
                    scanf("%d",p+i);
            for(i=0;i<n;i++)
                    scanf("%d",q+i);
            for(i=0;i<m;i++)
                    *(r+i)=*(p+i);
            for(i=m;i<m+n;i++)
                    *(r+i)=*(q+i-m);
            for(i=0;i<m+n;i++)
            printf("\t%d",*(r+i));
            free(p);
            free(q);
            free( r);
}
```

Example 7.3.6: Write a program that reads 'n' elements of a one dimensional array dynamically and counts the prime numbers present in it using function. [2063/p/9]

```
#include<stdio.h>
void prime_count(int*,int);
void main()
{
    int n,*address,i;
    printf("Enter Number of elements:");
    scanf("%d",&n);
    address=(int*)calloc(n,sizeof(int));
    if(n==NULL)
            exit();
    printf("Enter elements:");
    for(i=0;i<n;i++)
    {
            scanf("%d",(address+i));
    }
    prime_count(address,n);
}
void prime_count(int *array,int k)
{
    int i,j,count=0;
    for(i=0;i<k;i++)
    {
            for(j=2;j<*(array+i);j++)
            {
                    if(*(array+i)%j==0)
                            break;
            }
            if(j==*(array+i))
                    count=count+1;
    }
    printf("No. of prime numbers is %d", count);
}
```

```
Enter Number of elements: 5
Enter elements: 34 19 77 31 45
No. of prime numbers is 2
```

7.3.6 Two-dimensional dynamic memory allocation

Two-dimesional dynamic memory allocation is similar to one dimensional dynamic memory allocation. We should know that a two dimensional array is an array of one dimensional arrays. For two dimensional memory allocations, it is required to allocate memory for an array of pointers. Each element of the array holds the starting address of each row of the two dimensional array. If we want to allocate memory for a two dimensional array of size rows X cols, we have to declare a pointer variable that points to pointers. For example, let us take an example to allocate memory for a 4 X 4 matrix. Let us declare an identifier matrix as a pointer variable that points to an array of pointers. Each member of the array points to each array of the 4X4 array in one to one fashion.

```
float **matrix ;
```

where matrix is a pointer to point to an array of float pointers. To create an array of pointers of size 4, calloc function can be used as in the following syntax:

```
matrix=(float**)calloc(sizeof(float*), 4);
```

Here, calloc function allocates memory for an array of size 4 and returns the base address (starting address) to matrix. (1930 in figure 7.3.4). To allocate memory for each row of the array, calloc function can be used again as shown in the following code segment:

```
for(i=0;i<4;i++)
{
        matrix[i]=(float*)calloc(sizeof(float), 4);
}
```

Here, in the first round of loop, calloc function allocates memory for the first row and returns the base address to matrix[0] (1942 in figure 7.3.4). Similarly for second, third and fourth rows.
This concept is vidualized clearly in figure 7.3.4.

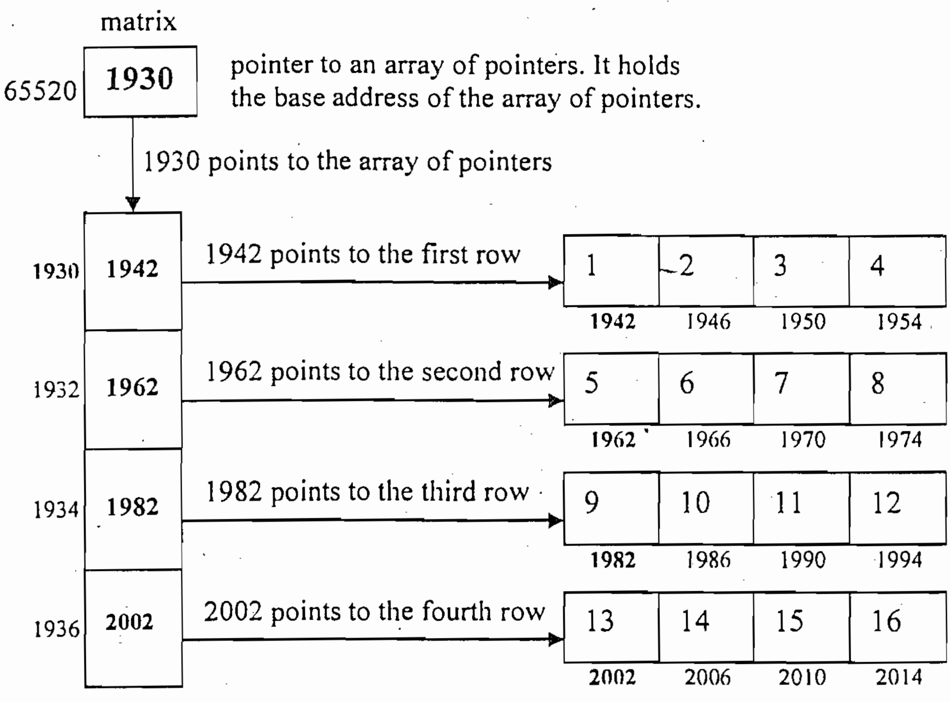


Fig. 7.3.4 Illustration of 2D dynamic memory allocation for a matrix of size 4X4

Question: How to access array element in fourth row and third column.
Answer:

- Address of array element that stores the address of the fourth row is (matrix+3)=(1930+3*scale factor of float pointer) = (1930+3*2)= 1936
- Value at (matrix+3) is *(matrix+3) = 2002 (base address of fourth row).
- Address of element at fourth row and third column is (*(matrix+3)+2) = (2002+2*4) = 2010
  [Here, 4 is the scale factor of float data type]
- Similarly, value at (*(matrix+3)+2) is *(*(matrix+3)+2) = 15.

Similarly, address of element at i^{th} row and j^{th} column is (*(matrix+i)+j) value at the location pointed by the address is *(*(matrix+i)+j).

7.3.7. Some important examples

The following examples use the above-described memory management functions to allocate memory dynamically in two -dimensional arrays. After memory allocation, all the operation can be done as in two-dimensional arrays.

Example 7.3.7: An example to allocate memory for a two-dimensional dynamic array.

```
#include<stdio.h>
void main(void)
{
        int rows,cols,i,j;
        float **matrix;
```

```
        printf("Enter maximum rows:");
        scanf("%d",&rows);
        printf("Enter maximum columns:");
        scanf("%d",&cols);
        /* Allocating memory for the matrix.*/
        /* First allocating a single dimensional array of n float pointers. */
        matrix=(float**)calloc(sizeof(float*),rows);
        if(matrix==NULL)
                exit();
        /* Now for each row, allocating a single dimensional array of floats. */
        for(i=0;i<rows;i++)
        {
                matrix[i]=(float*)calloc(sizeof(float),cols);
                if(matrix[i]==NULL)
                        exit();
        }
        for(i=0;i<rows;i++)
        {
                for(j=0;j<cols;j++)
                {       printf("\nEnter element[%d][%d]",i,j);
                        /* scanf("%f",&matrix[i][j]); */
                        scanf("%f",(*(matrix+i)+j));
                }
        }
        for(i=0;i<rows;i++)
        {
                for(j=0;j<cols;j++)
                {       printf("%f",*(*(matrix+i)+j));
                }
        }

        free(matrix);

}
void linkfloat()
{       float x=0,*y;
        y=&x;
        x=*y;
}
```

If **linkfloat** function is not defined, we will get the error "Floating Point Formats Not Linked" with majority of C compilers. When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating point emulator. A floating point emulator is a component which is used to manipulate floating point numbers in runtime library functions like scanf() and atof(). It occurs mainly while reading a float in a structure, an array of structures and in some other case. **linkfloat** function forces linking of the floating point emulator into an application. This function is not required to call. It is sufficient to define it anywhere in the program.

Example 7.3.8: A program to use functions to read, process, and display the sum of two dimensional dynamic arrays.

```
#include<stdio.h>
void display(float**,int,int);
void readmatrix(float**,int,int);
void processmatrix(float**,float**,int,int);
void main(void)
{       int rows1,cols1,i,j,rows2,cols2;
        float **matrix1,**matrix2;
        printf("Enter maximum rows of matrix1:");
        scanf("%d",&rows1);
        printf("Enter maximum columns of matrix1:");
        scanf("%d",&cols1);
        printf("Enter maximum rows of matrix2:");
        scanf("%d",&rows2);
        printf("Enter maximum columns of matrix2:");
        scanf("%d",&cols2);
        /* Allocating memory for the matrix1, matrix2. */
        /* First allocating a single dimensional array of n float pointers.*/
        matrix1=(float**)calloc(sizeof(float*),rows1);
        matrix2=(float**)calloc(sizeof(float*),rows2);
        /* Now for each row,allocating a single dimensional array of floats. */
        for(i=0;i<rows1;i++)
        {
                matrix1[i]=(float*)calloc(cols1,sizeof(float));
        }
        for(i=0;i<rows2;i++)
        {
                matrix2[i]=(float*)calloc(cols2,sizeof(float));
        }
        readmatrix(matrix1,rows1,cols1);
```

```
        readmatrix(matrix2,rows2,cols2);
        display(matrix1,rows1,cols1);
        printf("\n");
        display(matrix2,rows1,cols1);
        processmatrix(matrix1,matrix2,rows1,rows2);
        printf("\n");
        display(matrix1,rows1,cols1);
        free(matrix1);
        free(matrix2);
}
void display(float **mat, int r1,int c1)
{
        int i,j;
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                {       printf("%f ",*(*(mat+i)+j));
                }
        }
}
void readmatrix(float **mat,int r1,int c1)
{
        int i,j;
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                {
                        printf("Enter element [%d][%d]",i,j);
                        scanf("%f",(*(mat+i)+j));
                }
        }
}
void processmatrix(float** mat1,float** mat2,int r1,int c1)
{
        int i,j;
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                {
                        *(*(mat1+i)+j)=*(*(mat1+i)+j)+*(*(mat2+i)+j);
                        /* mat1[i][j]=mat1[i][j]+mat2[i][j]; */
                }
        }
}
```

Output

```
Enter maximum rows of matrix1:2
Enter maximum colunmns of matrix1:2
Enter maximum rows of matrix2:2
Enter maximum colunmns of matrix2:2
Enter element [0][0]        1       Enter element [0][1]    2
Enter element [1][0]        3       Enter element [1][1]    4
Enter element [0][0]        5       Enter element [0][1]    6
Enter element [1][0]        7       Enter element [1][1]    8
1.000000 2.000000 3.000000 4.000000
5.000000 6.000000 7.000000 8.000000
6.0      8.000000 10.000000 12.000000
```

Exercise 7.3

1. Differentiate between compile time and run time.
2. What is memory leak?
3. Why does the size of heap keep changing?
4. What do you understand by dynamic memory allocation? Explain the functions used for dynamic memory allocation in details with examples. [062/p/-9, 058/c/6-a, 058/p/5-a]
5. What type of values do memory allocation functions return? [059/p/6-a]
6. Write briefly about Dynamic Memory Allocation process? [2057/c/5-a, 059/p/6-a]
7. What are the advantages of dynamic memory allocation? [060/p/5-a]
8. Explain with example the difference between calloc( ) and malloc( ) in terms of the function they perform. [2063/p/5]
9. Write a program that reads 'n' from the user and allocates memory to hold 'n' floating point numbers. User enters the numbers and your program should find the largest, smallest and mean of the numbers.[059/c/8]
10. Write a program that allocates the memory space as required by the user for three arrays. User enters the numbers for two arrays and the program sums the corresponding array elements and store them in the third array. [059/p/6-b]

    hint:
    ```
    int *p,*q,*sum,i;
    ..............................................
    p=(int *)calloc(n,sizeof(int));
    q=(int *)calloc(n,sizeof(int));
    sum=(int *)calloc(n,sizeof(int));
    ....................................................
    for(i=0;i<n;i++)
    *(sum+i)=*(p+i)+*(q+i);
    ....................................................
    ```
11. Write a computer program to read the score of 24 students in a class using float type of array. Find and display the average score of that class. Use the concept of function and pointer. [058 p/6-b]
12. Write a program to read 'n' numbers dynamically and sort it in descending order using functions. [2063/b/9]
13. Write a program to pass a two dimensional dynamic array to a function. The function raises the power of each element by 2.
14. Write all the programs in exercise 7.1 using dynamic memory allocation.

Chapter 7.4
Strings

7.4.1 Introduction
In C, a string is stored as a null-terminated array of characters. This means that after the last truly usable char there is a null, which is represented in C by '\0'. The subscripts used for the array start with zero (0). Any group of characters (except double quote sign) defined with in double quotation marks is a constant string.
Example: " We are very laborious students."
Character strings are used to build meaningful and readable programs. The operations performed on character strings are:

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of a string

7.4.2 Declaring string variables
A string variable is any valid C variable name and is always declared as array. The general form of declaration of a string is

char string_name[size];

Example- The following line declares a char array called **str**.
char str[14];
For this array, C provides fourteen consecutive bytes of memory. Only the first thirteen bytes are usable for character storage, because one byte must be used for null character to terminate the sting.
When a compiler assigns a character string to a character array, it automatically supplies a ***null*** character('\0') at the end of the string. Therefore, the size should be the maximum numbers of characters in the string plus one. String is stored in memory as ASCII codes of the characters that make up the string appended with 0 (ASCII value of null). If the string "Hello, world!" is stored in array **str**, the representation of the string will be done as shown in Figure 7.4.1.
The name of the array is treated as a pointer to the array. The subscript serves as the offset into the array i.e., the number of bytes from the starting memory location of the array. Thus, both of the following will save the address of the 0th character in the pointer variable **ptr**.

Characters	H	e	l	l	o	,		w	o	r	l	d	!	'\0'
ASCII	72	100	107	107	111	44	32	119	111	114	107	100	33	0
Subscripts	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Fig.7.4.1 representation of a string in memory

```
ptr = str;
ptr = &str[0];
```

7.4.3 Initializing string variables
Character array can be initialized when they are declared. C permits a character array to be initialized in either of the following two forms.

```
char name[15]="kathmandu city";
char name[15]={'k','a','t','h','m','a','n','d','u',' ','c','i','t','y','\0'};
```

C also permits to initialize the character array without specifying the numbers of elements as-

```
char name[]="kathmandu city";
char name[]={'k','a','t','h','m','a','n','d','u',' ','c','i','t','y','\0'};
```

7.4.4 Reading strings from terminal

7.4.4.1 Reading words
The **scanf** function can be used with %s format specification to read a string of characters. The syntax is as

```
char name[15];
scanf("%s",name);
```

The problem with the scanf function is that it terminates its input when the first white space it finds.
Note that unlike other variable's scanf calls, in case of character arrays, the ampersand(&) is not required before the variable name because array name itself is the address of first byte of allocated space. (see chapter 7.4) The scanf function automatically terminates the read string with a **null** character. If we want n numbers of words then we need to read them n times.

Example 7.4.1: This program reads and displays a single word using scanf function.

```
#include<stdio.h>
void main()
{
        char namestr[10];
        printf("Enter name:");
        scanf("%s",namestr);
        printf("Name: %s",namestr);
}
```

7.4.4.2 Reading a line of text

Sometimes, we need to read an entire line of text from the terminal. At that time **scanf** function does not work. In such situations, we need to read the string character by character by using function **getchar** in loop and assigning to the character array until a certain condition is met. After terminating the reading, the null character should be inserted at the end of the string.

Example 7.4.2: This example reads a string character by character until enter key is pressed (new line character (\n)) and inserts the null character at the end. Instead of new line other character can be used. This is a technique to read strings having white space characters.

```
#include<stdio.h>
void main()
{           char namestr[100],ch;
            int i;
            printf("Enter any string:");
            i=0;
            do
            {           ch=getchar();
                        namestr[i]=ch;
                        i++;
            }while(ch!='\n');
            i=i-1;
            namestr[i]='\0';
            printf("String: %s",namestr);
}
```

7.4.5 Writing strings to the screen

printf function with %s format specifications is used to print the strings to the screen. It is discussed in detail in chapter 4.

Example 7.4.3: Study and understand the following example

```
#include<stdio.h>
void main()
{           char namestr[100]="kathmandu";
            int i=0,j;
            do
            {   j=0;
                        while(j<=i)
                        {
                                    printf("%c",namestr[j]);
                                    j++;
                        }
                        printf("\n");
                        i++;
            }while(namestr[i]!='\0');
}
```

Output of example 7.4.3

```
k
ka
kat
kath
kathm
kathma
kathman
kathmand
kathmandu
```

7.4.6 Arithmetic operations on characters

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. To write a character in its integer representation, we can write it as an integer. For example, suppose

```
x='A';
printf("%d" ,x);
```

will display the number 65 on the screen. It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x='A'+5;
```

is valid statement and value of x will be 70.

C also allows us also to use the character constants in relational expressions. for example, the expression **ch>='A'&&ch<='Z'** would test whether the character contained in the variable ch is an upper case letter. A digit character can be converted in to its equivalent integer value using the relationship: **i=character-'0';** Where i is an integer variable and character contains the character digit. Let us suppose that character contains the digit '8', Then,

i =ASCII value of '8'-ASCII value of '0'
=56-48
=8

The library functions **atoi converts a string of digits into their integer values**. The function takes the following form.

i=atoi(digit_string);

i is an integer variable and digit_string is a character array of digits. Both of the above concept is illustrated in example 7.4.4.

Example 7.4.4: This example shows to get the equivalent integer value of a character digit and the equivalent integer value of string of character digits.

```
#include<stdio.h>
void main()
{	char ch[5];
	char character;
	int n,x;
	printf("Enter a digit character:");
	character=getche();
	n=character-'0';
	printf("\nThe equivalent value is %d",n);
	printf("\nEnter a string of digit characters:");
	scanf("%s",ch);
	x=atoi(ch);
	printf("\nEquivalent number is %d",x);
}
```

Output

```
Enter a digit character:4
The equivalent value is 4
Enter a string of digit characters:56
Equivalent number is 56
```

7.4.7 String handling functions

We need to manipulate strings in different ways. As for example, concatenating two strings, copying one string to another, comparing two strings, finding the length of a string. To do so we have provided different types of library functions. These library functions are declared in **<string.h>**. The functions are briefly discussed below.

7.4.7.1 strlen()

This function finds the length of a string. To find the length of string, it counts and returns the number of characters in the string without including the null character. The general form is

l=strlen(string);

where l is the integer variable which receives the value of the length of the string.

string is an argument given to find the length. Which may be a constant string also.

Example 7.4.5: Illustration of strlen().

```
#include<stdio.h>
#include<string.h>
void main()
{
		char text1[100];
		int length;
		printf("Enter a string:");
		gets(text1);
		length=strlen(text1);
		printf("Length of the given string=%d",length);
}
```

Same task can be done using user defined function **udstrlen**, which is illustrated in example 7.4.6.

Example 7.4.6: Finding length of a given string.

```
#include<stdio.h>
int udstrlen(char[]);
void main(void)
{
	char text1[205];
	int length;
	printf("Enter a string:");
	gets(text1);
	length=udstrlen(text1);
	printf("Length of given string is : %d",length);
	getch();
}
int udstrlen(char txt1[])
{
	int len;
	for(len=0; txt1[len]!='\0'; len++);
	return len;
}
```

Output

```
Enter a string:Programming is fun.
```

7.4.7.2 strcat()

This concatenates(joins) two strings together. Its general form is:

strcat(string1,string2);

string1 and **string2** are character arrays. Which are given as argument to the function. When strcat is executed ,string2 is appended to string1. It does so by removing the null character at the end of string1 and placing string2 from there. The string2 remains unchanged. Which is illustrated in example 7.4.7.

Example 7.4.7: Illustration of strcat().

```
#include<stdio.h>
void main()
{
        char text1[100];
        char text2[50];
        printf("Enter first string1:");
        gets(text1);
        printf("Enter second string2:");
        gets(text2);
        strcat(text1,text2);
        puts(text1);
}
```

Same task can be done using user defined function **udstrcat**, which is illustrated in example 7.4.8.

Example 7.4.8: Concatenating two strings.

```
#include<stdio.h>
void udstrcat(char[],char[]);
void main(void)
{
        char text1[205];
        char text2[100];
        printf("Enter string1:");
        gets(text1);
        printf("Enter string2:");
        gets(text2);
        udstrcat(text1,text2);
        puts(txt1);
}
void udstrcat(char txt1[],char txt2[])
{
        int i,j;
        for(i=0; txt1[i]!='\0'; i++);
        for(j=0; txt2[j]!='\0'; i++, j++)
        {
                txt1[i]=txt2[j];
        }
        txt1[i]='\0';
        printf("Concatenated string is:");
}
```

Output

```
Enter string1:FIRST
Enter string2:LAST
Concatenated string is:FIRSTLAST
```

7.4.7.3 strcpy()

This function copies one string to another. It takes the form:

strcpy(deststring,sourcestring);

deststring and **sourcestring** are two strings as argument to the function. Which may be character array or string constant. When this function is executed, it will assign the content of string variable **sourcersting** to the string variable **deststring.**

Example 7.4.9: Illustration of strcpy().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char text1[100];
        char text2[50];
        printf("Enter first string1:");
        gets(text1);
        printf("Enter second string2:");
        gets(text2);
        strcpy(text1,text2);
        puts(text1);
}
```

The same task can be done without using **strcpy**, which is illustrated in example 7.4.10.

Example 7.4.10: Copying one string to another.

```
#include<stdio.h>
void udstrcpy(char[],char[]);
void main()
{
        char text1[50];
        char text2[50];
        printf("Enter first string1:");
        gets(text1);
        printf("Enter second string2:");
        gets(text2);
        udstrcpy(text1,text2);
        printf("%s",text1);
}
void udstrcpy(char txt1[],char txt2[])
{
        int i;
        i=0;
        while(txt2[i]!='\0')
        {
                txt1[i]=txt2[i];
                i++;
        }
        txt1[i]='\0';
}
```

Output

```
Enter first string1:washington DC
Enter second string2:Kathmandu
Kathmandu
```

7.4.7.4 strcmp()

The strcmp function compares two strings. The general form of the string is:

strcmp(string1,string2);

Where string1 and string2 are arguments, which may be character arrays or string constants. When this function is executed it returns 0 if both strings are equal. If the strings are not equal, it returns the numeric difference between first mismatching characters in the strings.

Example 7.4.11: Illustration of strcmp().

```
#include<stdio.h>
#include<string.h>
void main()
{
        char text1[100];
        char text2[50];int c;
        printf("Enter first string:");
        gets(text1);
        printf("Enter second string:");
        gets(text2);
        c=strcmp(text1,text2);
        if(c==0)
                printf("Strings are equal.");
        else
                printf("Strings are not equal.");
}
```

The same task can be done without using the **strcpy** which is illustrated in example 7.4.12.

Example 7.4.12: Comparing two strings.

```
#include<stdio.h>
int udstrcmp(char[],char[]);
void main()
{
        char text1[50];
        char text2[50];
        int dif;
        printf("Enter first string:");
        gets(text1);
        printf("Enter second string:");
        gets(text2);
        dif=udstrcmp(text1,text2);
        if(dif==0)
                printf("\nDifference is %d.So,strings are equal.",dif);
        else
                printf("\nDifference is %d.So,strings are unequal.",dif);
}
int udstrcmp(char txt1[],char txt2[])
```

```
        int i=0,d;
        do
        {
                d=txt1[i]-txt2[i];
                if(d!=0)
                        break;
                i++;
        }while(txt1[i]!='\0'||txt2[i]!='\0');
        return d;
}
```

Output

Enter first string:aaaa
Enter second string:aaaaa
Differenec is -97.So,strings are unequal.

7.4.8 Arrays of strings (table of strings)

Generally, we use lists of character strings, as for example, name of students in a class, lists of names of employs in an organization etc. A list of name can be treated as a table of strings and a two dimensional character array can be used to store the entire list. For example, a character array **namelist[5][20];** may be used to store a list of 5 names, each of length not more than 20 characters.

Suppose, if we initialize the above array as:

char namelist[5][20]={"Khagendra", "Amit", "Khushbu", "Neelu", Sudarshan"};

Which is stord in the tabular form as shown in the following figure:

K	h	a	g	e	n	d	r	a	'\0'			
A	m	i	t	'\0'								
K	h	u	s	h	b	u	'\0'					
N	e	e	l	u	'\0'							
S	u	d	a	r	s	h	a	n	'\0'			

Fig. 7.4.2 Representation of arrays of strings

Example 7.4.13 shows how to input, output and access elements of 2D-arrays of characters

Example 7.4.13: Sorting of strings in ascending order.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char namelist[5][40],temp[40]; int pass,j;
        printf("Enter names:");
        for(j=0;j<5;j++)
        gets(namelist[j]);
        for(pass=0;pass<5-1;pass++)
        {
                for(j=0;j<5-pass-1;j++)
                {
                        if(strcmp(namelist[j],namelist[j+1])>0)
                        {
                                strcpy(temp,namelist[j]);
                                strcpy(namelist[j],namelist[j+1]);
                                strcpy(namelist[j+1],temp);
                        }
                }
        }
        printf(Sorted  name list is:);
        for(j=0;j<5;j++)
        puts(namelist[j]);
}
```

Output

Enter names:
Pankaj Laminechhane
Pratik Gauchan
Prakash Dangol
Arun Shrestha
Santosh Prajapati

Sorted name list is:
Arun Shrestha
Pankaj Laminechhane
Prakash Dangol
Pratik Gauchan
Santosh Prajapati

7.4.9 Strings and Pointer

Like in one dimensional arrays, we can use a pointer to access the individual characters in a string. Which is illustrated in example 7.4.14 and 7.4.15.

Example 7.4.14: Reading and writhing strings using pointer notation.

```
#include<stdio.h>
void main()
{        char *text;
         printf("Enter a string:");
         gets(text);
         puts(text);
}
```

Example 7.4.15: Reading and writing strings using pointer notation until + is encountered. This technique is used to read strings having different types of white space characters.

```
#include<stdio.h>
void main()
{
         char *text;
         printf("Enter a string:");
         scanf("%[^+]",text]);
         printf("%s",text);
}
```

Output of example 7.4.15

```
Enter a string:Kathmandu Engineering College
is located at kalimati.+
Kathmandu Engineering College
is located at kalimati
```

Where text is a pointer to a character and assigns the address of first character as the initial value.
All above discussed examples can be done using pointer notation. One important use of pointers is in handling of a table of strings. Suppose the following array of strings:

char namelist[3][20];

This says that namelist is a table containing three names, each with a maximum length of 20 characters(including null character). The total storage requirements for the namelist table are 60 bytes. We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row of a fixed number of characters, we can make it pointer to a string of varying length.
For example,

char *namelist[3]={"Shakti Kiran","Bishow", "Atul"};

declare array of pointers **namelist** of type char and size 3. Each pointer points each row of 2D array of characters i.e., each pointer pointing to a particular name as shown below:

namelist[0] ⟶ Shakti Kiran
namelist[1] ⟶ Bishow
namelist[2] ⟶ Atul

This declaration allocates only 25 bytes, sufficient to hold all characters as shown below.

S	h	a	k	t	i		K	i	r	a	n	'\0'
B	i	s	h	o	w	'\0'						
A	t	u	l	'\0'								

Fig. 7.4.3 representation of arrays of strings using pointer notation

Example 7.4.16 shows sorting of arrays of string using pointer. In example 7.4.13, to sort string we copied the string from one string to another string but in example 7.4.16 we will swap the pointers. This technique will eliminates:

- Complicated storage management
- High overheads of moving lines

Example 7.4.16: Sorting name list using concept of pointer and function.

```
#include<stdio.h>
void namelistsort(char**);
void main()
{
         char *namelist[5],*temp;
         int i;
         printf("\nEnter any five strings:");
         for(i=0;i<5;i++)
         {
                  gets(namelist[i]);
         }
         namelistsort(namelist);
         printf("\nSorted name list is:\n");
         for(i=0;i<5;i++)
         {
                  puts(namelist[i]);
```

```
}
void namelistsort(char *namelist[])
{
	char *temp;
	int i,j,pass;
	for(pass=0;pass<5-1;pass++)
	{
		for(j=0;j<5-pass-1;j++)
		{
			if(strcmp(namelist[j],namelist[j+1])>0)
			{
				temp=namelist[j];
				namelist[j]=namelist[j+1];
				namelist[j+1]=temp;
			}
		}
	}
}
```

Output

```
Enter any five strings:
Mukesh Regmi
Padam GC
Niraj Shakaya
Baijantee Rajbashi
Grishma Kaphle

Sorted name list is:
Baijantee Rajbashi
Grishma Kaphle
Mukesh Regmi
Niraj Shakaya
Padam GC
```

In the above example, the number of rows are fixed. But we can make array of strings (2D array of characters) as illustrated in example 7.4.17.

Example 7.4.17: Illustration of two-dimensional dynamic array of characters.

```
#include<stdio.h>
void main()
{
	char **namelist; int n,i;
	printf("Enter numbers of items:");
	scanf("%d",&n);
	fflush(stdin); /* To flush buffer (see chapter 9)*/
	for(i=0;i<n;i++)
	{
		gets(namelist[i]);
	}
	for(i=0;i<n;i++)
	{
		puts(namelist[i]);
	}
}
```

7.4.10 Character conversions and testing

We conclude this chapter with a related library #include <ctype.h> which contains many useful functions to convert and test single characters. The common functions prototypes as follows:

7.4.10.1 Character testing functions

Each of the following functions takes a character type argument as an argument and returns a non zero (true) integer and zero (false) value.

Table 7.4.1 Some important functions in <ctype.h>		
Return type	**Function**	**Returns**
int	isalnum(c)	True if c is alphanumeric.
int	isalpha(c)	True if c is a letter.
int	isascii(c)	True if c is ASCII.
int	iscntrl(c)	True if c is a control character.
int	isdigit(c)	True if c is a decimal digit.
int	isgraph(c)	True if c is a graphical character.
int	islower(c)	True if c is a lowercase letter.
int	isprint(c)	True if c is a printable character.
int	ispunct(c)	True if c is a punctuation character.
int	isspace(c)	True if c is a space character.
int	isupper(c)	True if c is an uppercase letter.
int	isxdigit(c)	True if c is a hexadecimal digit.

For example, if the given character to the **isupper** is uppercase then **isupper** returns non-zero value (true) otherwise it returns 0.

7.4.10.2 character conversion functions

int toascii(c)-converts value of argument (c) to ASCII.
int tolower(c)-converts given argument (c) to lowercase.
int toupper(c)-converts a character (c) to uppercase.
Where c is character type argument. All the functions have **int** return type.

Example 7.4.18: A program to illustrate the concept of different character conversion and testing functions.

```
#include<stdio.h>
void main()
{
        int i;
        char c;
        printf("Enter a alphabetic lowercase character:  ");
        c=getchar();
        putchar(toupper(c));
        fflush(stdin);
        printf("\nEnter an alphabetic character:");
        c=getchar();
        if(islower(c))
                printf("%c is a lowercase character.",c);
        else
                printf("%c is a uppercase character.",c);
}
```

Output of example 7.4.18

```
Enter an alphabetic lowercase character:  m
M
Enter a alphabetic character: h
h is a lowercase character
```

7.4.19 A program to convert lower case characters to upper case and vice versa in a string entered by a user and finding the number of words in the string.

```
#include<stdio.h>
#include<ctype.h>
void main()
{
        char name[205];
        char ch;
        int count=0,j,i=0;
        printf("Enter a string having lower case and upper case charactres terminated with * ");
        do
        {
                ch=getchar();
                if(ch=='*')
                {
                        name[i]='\0';
                }
                else
                {
                        name[i]=ch;
                        if(name[i]==' ')
                        {
                                count++;
                        }
                }
                i++;
        }while(ch!='*');
        for(j=0;j<i;j++)
        {
                if(islower(name[j]))
                        name[j]=toupper(name[j]);
                else
                        name[j]=tolower(name[j]);
        }
        for(j=0;j<i;j++)
        {
                printf("%c",name[j]);
        }
        printf("\nTotal number of words are %d",count+1);
}
```

```
Enter a string having lower case and upper case charactres terminated with *
OCEN CAN DRY, mountain can fly, BUT LOVE CAN NOT DIE*
ocen can dry, MOUNTAIN CAN FLY, but love can not die
```

7.4.11 Some important examples

Example 7.4.20: Write a program that takes a string and passes to a function. The function takes the string and converts all of its lower case characters to upper case characters if the first character is in lower case and vice versa. Display the string before conversion and after conversion in the calling function. Your function must convert the original string not a copy of it. [062/b/6-b]

```
#include<stdio.h>
void stringconvert(char[]);
void main()
{
        char anystring[100];
        printf("Enter a string:");
        gets(anystring);
        printf("String before conversion : ");
        puts(anystring);
        stringconvert(anystring);
        printf("\nstring after conversion : ");
        puts(anystring);
}
void stringconvert(char anystr[])
{
        int i;
        if(anystr[0]>='a'&&anystr[0]<='z')
        {       i=0;
                while(anystr[i]!='\0')
                {
                        if(anystr[i]>='a'&&anystr[i]<='z')
                        {
                                anystr[i]=anystr[i]-32;

                        }

                        i++;
                }
        }
        else if(anystr[0]>='A'&&anystr[0]<='Z')
        {       i=0;

                while(anystr[i]!='\0')
                {
                        if(anystr[i]>='A'&&anystr[i]<='Z')
                        {
                                anystr[i]=anystr[i]+32;
                        }
                        i++;


                }
        }
        else
        {
                printf("\nString is started wothout a upper case or a lower case.");
        }
}
```

Example 7.4.21: Example 7.4.20 using pointer notation.

```
#include<stdio.h>
void stringconvert(char*);
void main()
{

        char *anystring;
        printf("Enter a string:");
        gets(anystring);
        printf("String before conversion : ");
        puts(anystring);
        stringconvert(anystring);
        printf("\nstring after conversion : ");
        puts(anystring);
}
void stringconvert(char *anystr)
{
        int i;
        if(*(anystr+0)>='a' && *(anystr+0)<='z')
        {
```

```
            i=0;
            while(*(anystr+i)!='\0')
            {
                    if(*(anystr+i)>='a' && *(anystr+i)<='z')
                    {
                            *(anystr+i) = *(anystr+i)-32;
                    }
                    i++;
            }
    }
    else if(*(anystr+0)>='A' && *(anystr+0)<='Z')
    {
            i=0;
            while(*(anystr+i)!='\0')
            {
                    if(*(anystr+i)>='A' && *(anystr+i)<='Z')
                    {
                            *(anystr+i) = *(anystr+i)+32;
                    }
                    i++;
            }
    }
    else
    {
            printf("\nString is started wothout a upper case or a lower case.");
    }
}
```

Example 7.4.22: Write a program to insert a string into another string in the location specified by the user. Read the strings in the main program and pass them to function along with the inserting position and return the resulting string. [062/p/12, 2063/p/10]

```
#include<stdio.h>
void insertstring(char[],char[],int);
void main()
{
        char mainstring[100],stringtobeinserted[20];
        int n;
        printf("Enter the main string:");
        gets(mainstring);
        printf("Enter the string to be inserted:");
        gets(stringtobeinserted);
        printf("Enter position to insert the string:");
        scanf("%d",&n);
        insertstring(mainstring,stringtobeinserted,n);
        printf("The new string is:");
        puts(mainstring);
}
void insertstring(char mstr[],char str[],int p)
{
        int i=p-1,j=0;
        char temp[100];
        while(mstr[i]!='\0')
        {
                temp[j]=mstr[i];
                i++;j++;
        }
        temp[j]='\0';j=0;i=p-1;
        while(str[j]!='\0')
        {
                mstr[i]=str[j];
                i++;j++;
        }
        j=0;
        while(temp[j]!='\0')
        {
                mstr[i]=temp[j];
                i++;j++;
        }
        mstr[i]='\0';
}
```

Output of example 7.4.22

```
Enter the main string: Kathmandu  Engineering.
Enter the string to be inserted: College
```

The new string is: Kathmandu College Engineering.

Example 7.4.23: A program to print the reverse of the given word recursively.

```
#include<stdio.h>
void rev(char[],int);
void main()
{
        char string[100];
        printf("Enter any sting:");
        gets(string);
        rev(string,strlen(string));
}
void rev(char s[],int l)
{       putchar(s[l]);
        if(l>=0)
        {
                rev(s,l-1);
        }
}
```

Example 7.4.24: Check whether a string entered by user is a palindrome or not and display a proper message. [Hint: A palindrome word spells same back and forth e g : madam, noon].

```
#include<stdio.h>
#include<string.h>
void main(void)
{
        char astring[100];
        int i,length,j,flag=1;
        printf("Enter a string:");
        scanf("%[^\n]",astring);
        length=strlen(astring);
        for(i=0,j=length;i<length/2;i++,j--)
        {
                if(astring[i]!=astring[j-1])
                {
                        flag=0;
                        break;
                }
        }
        if(flag==1)
                printf("\n%s is a palindrome",astring);
        else
                printf("\n%s is not a palindrome",astring);
}
```

Example 7.4.25: Write a program that prompts the user for two strings assemble them in to a single string and then print the string reversed. Check whether the assembled string is palindrome or not.

```
#include<stdio.h>
#include<string.h>
void main(void)
{
        char first[30];
        char second[40];
        char assembled[100];
        char reversed[100];
        int i=0,j=0,k=0;
        printf("Enter first string:");
        gets(first);
        printf("Enter second string:");
        gets(second);
        while(first[i]!='\0')
        {
                assembled[i]=first[i];
                i++;
        }
        for(;second[j]!='\0';)  /* we can use while(second[j]!='\0') */
        {
                assembled[i]=second[j];
                j++;i++;
        }
        assembled[i]='\0';
        puts(assembled);
        for(j=i-1;j>=0;j--)
        {
                reversed[k]=assembled[j];
                k++;
        }
        reversed[k]='\0';
```

```
    printf("The reversed string is:");
    puts(reversed);
    if(strcmp(assembled,reversed)==0)
        printf("The assembled string is palindrome");
    else
        printf("The assembled string is not palindrome");
}
```

Example 7.4.26: A program to find the frequency of a character in the string entered by a user.

```
#include<stdio.h>
#include<string.h>
void main(void)
{
    char astring[100],ch;
    int i=0,charactercount=0;
    printf("%c",ch);
    printf("Enter a string:");
    gets(astring);
    printf("Enter a character to find check its frequecy:");
    scanf("%c",&ch);
    do
    {
        if(astring[i]==ch)
        charactercount++;
        i++;
    }while(astring[i]!='\0');
    printf("\nThe number of %c is %d",ch,charactercount);
}
```

Example 7.4.27: A program to count the frequency of characters in the string entered by a user.

```
#include<stdio.h>
void main(void)
{
    char string[100];
    int counts[256]={0},i=0,j;
    printf("Enter a string:\n");
    scanf("%[^\n]",string);
    while(string[i]!='\0')
    {
        counts[string[i]]=counts[string[i]]+1;
        i++;
    }
    for(j=0;j<256;j++)
    {
        printf("\t%c=%d",j,counts[j]);
    }
}
```

Example 7.4.28: Write a program that finds the longest word in a given string.

```
#include<stdio.h>
#include<string.h>
void main(void)
{
    char string[200],tempword[50],word[50];int i=0,length=0,count=0;
    printf("Enter a string:");
    scanf("%[^\n]",string);
    while(string[i]!='\0')
    {   if(string[i]==' ')
        {
            tempword[count]='\0';
            if(length<count)
            {
                length=count;
                strcpy(word,tempword);
            }
            count=0;
        }
        else
        {
            tempword[count]=string[i];
            count++;
        }
        i++;
    }
    printf("The longest word is  %s and its length is %d.",word,length);
}
```

Output

Enter a string:
Nepal is a beautiful country

Example 7.4.29: A program to read a character and test whether it is an alphabet or a number or special character or control character.

```
#include<stdio.h>
void main(void)
{
	char ch;
	printf("Enter a charater:");
	ch=getche();
	printf("%d",ch);
	if(ch>=65&&ch<=90||ch>=97&&ch<=122)
		printf("The character is alphabet");
		else if(ch>=48&&ch<=57)
			printf("\nThe character is digit");
			else if(ch>=0&&ch<=32||ch==127)
				printf("\nThe character is control character");
					else
					printf("\nSpecial character");
}
```

Example 7.4.30: Write a program to read a string. Now pass this string to a function that finds the number of vowels, consonants, digits, white spaces (space and tab character) and other characters in the string. Your function must return these numbers to the calling function at once. In the calling function, display those numbers. [061/p/5-b]

```
#include<stdio.h>
#include<ctype.h>
void charactercount(char[],int[]);
void main(void)
{
	static char string[200];
	int counts[5]={0,0,0,0,0};
	printf("Enter a string:");
	gets(string);
	charactercount(string,counts);
	printf("\nNumber of vowels =%d",counts[0]);
	printf("\nNumber of consonants =%d",counts[1]);
	printf("\nNumber of digits =%d",counts[2]);
	printf("\nNumber of white space =%d",counts[3]);
	printf("\nNumber of other characters=%d",counts[4]);
}
void charactercount(char str[],int count[])
{
	int i;
	for(i=0;str[i]!='\0';i++)
	{
		if(isalpha(str[i])!=0)
		{
			if(str[i]=='a'||str[i]=='e'||str[i]=='i'||str[i]=='o'||str[i]=='u'||str[i]=='A'||str[i]=='E'||str[i]=='I'||str[i]=='O'||str[i]=='U')
				count[0]++;
			else
				count[1]++;
		}
		else if(isdigit(str[i])!=0)
			count[2]++;
		else if(isspace(str[i])!=0)
			count[3]++;
		else
			count[4]++;
	}
}
```

Output

```
Number of other characters=7
Enter a string:AA E II oo uuu klmn XYZ 4646546 ;'";]&*%
Number of vowels =10
Number of consonants =7
Number of digits =7
Number of white space =8
```

Exercise 7.4

1. Define string. How is the end of a string recognized in C?
2. What is main difference between an integer arrays and strings?
3. What is the difference between the declaration of a character and a string?
4. What are possible arithmetic operations on characters?
5. Write briefly about main four string handling functions with their input and output and examples.
6. How are two-dimensional strings are represented?
7. State some roles of character testing and conversion functions in C.
8. What is the role of string handling functions in program coding? Explain with few examples only [2057/c/3-a]
9. What is the importance of string handling functions in 'C'? Where do we use **strcat** and **strcmp** function.[058/p/3-a]

10. Write a program that reads a string and convert all uppercase characters to lowercase and vice versa and displays the string. [058/c/6-b, 057/c/3-b]
11. Write a program that takes a string and converts it touppercase if the first character is uppercase or to lowercase if the first character is lowercase. [059/p/4-b, 059/p/4-b, 059/c/2-b-similar, 060/p/5-b]
12. Write a program that takes a string from the user and pass it to a function. The function finds and returns the number of words in the string. Display the number of words in the main program.[061/c/4-b]
13. Write a program that reads your name from the keyboard and output a list of ASCII codes, which represent your name.
14. Write a program to extract a portion of a character string and print the extracted string. Assume that m characters are extracted, starting with the nth character.
15. Try to write all the programs in this chapter using for loops.
16. Write a program that will read a text and count all occurrences of particular words.
17. Write a program that will read a string and rewrite it in the alphabetical order. For example, the word NEPAL should be written as AELNP.
18. Discuss any five string handling functions that are not discussed in this chapter. [Hint: see help]
19. Write a program to read 20 names of students and sort them alphabetically. The process must be done by the user-defined function. [2063/b/10]
20. Write a program to read name, age and designation of 10 employs and display the information in tabular form.
21. Write a program to do the following:
    To print the question "Who is the prime minister of Nepal"?
    To accept the answer.
    To print "Good" and stop if the answer is correct.
    To print the message "try again", if the answer is wrong.
    To display the correct answer when the answer is wrong even at the third attempt and stop.

Chapter 8

8.1 Structures 8.2 Unions

use → when we used the structure.

8.1.1 Introduction

061B

A structure provides a means of grouping variables under a single name for easier handling and identification. It can be defined as a new named type, thus extending the number of available types. It is a heterogeneous user defined data type. It is also called constructed data type. It may contain different data types. Structure can also store non-homogenous data types into a single collection. Structure may contain pointers, arrays or even other structures other than the common data types (such as int, float, long int etc) It is a convenient way of grouping several pieces of related information together. Complex hierarchies can be created by nesting structures. Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

8.1.2 Structure Definition (Structure template declaration)

The genaral of a structure definition is as follows:

```
struct tag_name
{
        data_type   member1;
        data_type   member2;
        ..............   ..............
        ..............   ..............
        data_type   member n;
}
```

A structure definition creates a format that may be used to declare structure variables. Let us use an example to illustrate the process of definition and the creation of structure variables. Let us consider a student information system consisting name, roll number, section and average marks in final exam.

We can define a structure to hold this information as follows:

```
struct student
{
        char name[50];
        int roll;
        char sec;
        float marks;
};
```

The keyword **struct** declares a structure to hold the details if four fields: name, roll, sec and marks. These fields are called structure member or elements. Each member may belong to different data types. **student** is the name of the structure and is called the **structure** tag. The tag name may be used subsequently to declare variables that have the tag's structure. Note that above declaration has not declared any variables. It simply describes a format called template to represent information as shown in table 8.1.1. Normally structure definition appears at the beginning of the program file, before any variables or functions are defined. They may also appear before the main as a global definition and can be used by other functions as well.

Table 8.1.1 A sample template of a structure

	struct student
name	array of 50 characters
roll	integer
sec	character
marks	float

8.1.3 Structure variable declaration

We can declare structure variables using the tag name any where in the program. For example the statement

struct student s1, s2, s3;

declares s1,s2,s3 as variables of type **struct student**. Each of these variables has four members as specified by the template. Members of structures are not variables. They do not occupy any memory until they are associated with the structure variables. The memory allocatin for structures of type student is illustrated in figure 8.1.1.

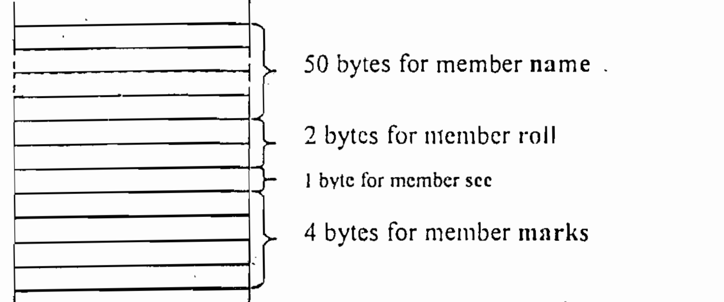


Fig. 8.1.1 Illustration of memory allocation while declaring structure variables of type student

We can combine both template declaration and variable declaration in one statement. The following declarations is also valid.

```
struct student
{       char name[50];
        int roll;
        char sec;
        float marks;
}s1,s2,s3;
```

8.1.4 Accessing members of a structure

There are two types of operators to access members of a structure. Which are:

- Member operator (dot operator or period operator(.))
- Structure pointer operator (->)

8.1.4.1 Member operator(.)

The link between member and variable is established using member operator when structure members are accessed by using structure variables (like s1,s2,s3 in above declaration). Any member of a structure can be accessed as:

structure_variable_name . manber_name

8.1.4.2 Structure pointer operator(->)

C provides the structure pointer operator -> to access members of a structure via a pointer. It is typed on the keyboard as a minus sign followed by a greater than sign. If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by:

pointer_to_structure -> member_name

Example 8.1.1: This example illustrates the above described concept.

```
#include<stdio.h>
void main(void)
{
        /* Structure definition at the starting of main function.*/
        struct student
        {
                char name[50];
                int roll;
                char sec;
                float marks;
        }; /* End of structure definition. */
        struct student s; /* Declaration of structure variable. Here s is a structure variable of type struct student */
        printf("Enter name of a student:");
        scanf("%s",s.name); /* Accessing member name of s using member operator. */
        printf("Enter roll:");
        scanf("%d",&s.roll); /* Accessing member roll of s using member operator. */
        printf("Enter section:");
        scanf(" %c",&s.sec); /* Accessing member sec (section) of s using member operator. */
        printf("Enter marks:");
        scanf("%f",&s.marks); /* Accessing member marks of s using member operator. */
        printf("name=%s\nRoll=%d\nSec=%c\nMarks=%f",s.name,s.roll,s.sec,s.marks);
}
```

8.1.5 Structure initialization

Like any data type, a structure variables can be initialized. The variable s of struct student can be initialized during its declaration as follows:

struct student s={"Radhika", 1234, 'A', 75.5};

The initial values for the components of the structure are placed with in curly braces and separated by commas.

8.1.6 Arrays of structures

Like any other data type variable, array of structure variables can be declared. The array will have individual structures as its elements. The array of structure is stored inside the memory in the same way as multi-dimensional array. Example 8.1.2 clarifies the concept of array of structures.

Example 8.1.2: This example declares an array of structure variable s having 3 members. Each member in turn is a structure having four members.

```
#include<stdio.h>
struct student
{       char name[50];
        int age;
        int roll;              Structure definition(global)
        char sec;
};
void main(void)
```

```
        int age;
};
struct student
{
        int roll;
        char sec;
        struct person p;
};
void main(void)
{       struct student s;
        printf("Enter name of a student:");
        scanf("%s",s.p.name); /* Accessing inner member. */
        printf("Enter age:");
        scanf("%d",&s.p.age);
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        scanf(" %c",&s.sec);
        printf("name=%s\nAge=%d\nRoll=%d\nSec=%c",s.p.name,s.p.age,s.roll,s.sec);
}
```

8.1.9 structures and functions

C supports passing of structures as arguments to functions. There are two methods by which the values of a structure can be transferred from one function to another.

- Passing by value
- Passing by reference

8.1.9.1 Passing by value

This method involves passing a copy of the entire structure to the called function. Any change to the structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function.

Example 8.1.5: In this example main function passes a structure to a function named display to display the structure members.

```
#include<stdio.h>
void display(struct student);
struct student
{       char name[50];
        int age;
        int roll;
        char sec;
};
void main(void)
{
        struct student s;int i;
        printf("Enter name of a student:");
        scanf("%s",s.name);
        printf("Enter age:");
        scanf("%d",&s.age);
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        scanf(" %c",&s.sec);
        display(s);
}
void display(struct student st)
{
        printf("name=%s\nAge=%d\nRoll=%d\nSec=%c",st.name,st.age,st.roll,st.sec);
}
```

8.1.9.2 Passing by reference

This method uses the concept of pointer to pass structure as an argument. The address location of the structure is passed to the called functions. The function can access indirectly the entire structure and work on it. This method is more efficient than the first one. Structure pointer operator is used to access the member.

Example 8.1.6: Example 8.1.5 is done again using pass by references. Study and understand the difference.

```
#include<stdio.h>
struct student
{       char name[50];
        int age;
        int roll;
        char sec;
```

```
void main(void)
{
        struct student s;int i;
        printf("Enter name of a student:");
        scanf("%s",s.name);
        printf("Enter age:");
        scanf("%d",&s.age);
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        scanf(" %c",&s.sec);
        display(&s);
}
void display(struct student *st)
{
        printf("name=%s\nAge=%d\nRoll=%d\nSec=%c", st->name,st->age,st->roll,st->sec);
}
```

Apart form above methods, structure members can be passed to functions individually.

Example 8.1.7: This example illustrates the concept of dynamic memory allocation for creating a array of structure of size n. In this example, a structure employee having members name, age and address is used. The array is passed to a function which sorts the array in descending order on the basis of member age and displays the sorted array from the main.

```
#include<stdio.h>
#include<alloc.h>
struct employee
{
        char name[50];
        int age;
        char address[50];
};
void sort(struct employee*,int);
void main(void)
{
        struct employee *p;
        int n,i;
        printf("Enter number of elements:");
        scanf("%d",&n);
        p=(struct employee*)calloc(n,sizeof(struct employee)); /* local function declaration */
        if(p==NULL)
                exit();
        for(i=0;i<n;i++)
        {       fflush(stdin);
                printf("Enter name:");
                gets((p+i)->name);
                printf("Enter age:");
                scanf("%d",&(p+i)->age);
                fflush(stdin);
                printf("Enter address:");
                gets((p+i)->address);
        }
        sort(p,n);
        for(i=0;i<n;i++)
        {
                printf("Name : %s\n",(p+i)->name);
                printf("Age : %d\n",(p+i)->age);
                printf("Address : %s\n",(p+i)->address);
        }
        free(p);
}
void sort(struct employee *pe,int num)
{       int pass=0,i;
        struct employee temp;
        for(pass=0;pass<num-1;pass++)
        {
                for(i=0;i<num-1-pass;i++)
                {
                        if((pe+i)->age<(pe+i+1)->age)
                        {
                                temp=*(pe+i);         /* structure assignment */
                                *(pe+i)=*(pe+i+1);    /* structure assignment */
                                *(pe+i+1)=temp;       /* structure assignment */
                        }
                }
        }
}
```

8.1.9 Some important examples

Example 8.1.8: Write a program to manipulate complex numbers using structures. The structure should contain real and imaginary part. Write separate functions to add and subtract complex numbers.
[062/b/7-b]

```
#include<stdio.h>
struct complexnum
{
        float real;
        float imag;
};
void add(struct complexnum,struct complexnum);
void subtract(struct complexnum,struct complexnum);
void main()
{
        struct complexnum n1,n2;
        printf("Enter real and inaginary part of first complex number:");
        scanf("%f%f",&n1.real,&n1.imag);
        printf("Enter real and inaginary part of second complex number:");
        scanf("%f%f",&n2.real,&n2.imag);
        add(n1,n2);
        subtract(n1,n2);
}
void add(struct complexnum nf,struct complexnum ns )
{
        float realsum,imagsum;
        realsum=nf.real+ns.real;
        imagsum=nf.imag+ns.imag;
        printf("The sum of complex numbers=%f+i(%f)\n",realsum,imagsum);
}
void subtract(struct complexnum nf,struct complexnum ns)
{
        float realdif,imagdif;
        realdif=nf.real-ns.real;
        imagdif=nf.imag-ns.imag;
        printf("\nThe sum of complex numbers=%f+i(%f)\n",realdif,imagdif);
}
```

Output

```
Enter real and inaginary part of first complex number:1 2
Enter real and inaginary part of second complex number:3 4
The sum of complex numbers=4.000000+i(6.000000)
The sum of complex numbers=-2.000000+i(-2.000000)
```

Example 8.1.9: Create a structure TIME containing hour, minutes and seconds as its member. Write a program that uses this structure to input start time and stop time to a function, which returns the sum and difference of the start time and stop time in the main program. [061/b/7-b, 061/p/7-b, 2063/p/11]

```
#include<stdio.h>
struct TIME
{
        int hour;
        int minutes;
        int seconds;
};
void display(struct TIME,struct TIME,struct TIME *,struct TIME *);
void main(void)
{
        struct TIME start,stop,sum={0,0,0},dif={0,0,0};
        printf("\nEnter hour,minutes and seconds of start time:");
        scanf("%d%d%d",&start.hour,&start.minutes,&start.seconds);
        printf("\nEnter hour,minutes and seconds of stop time:");
        scanf("%d%d%d",&stop.hour,&stop.minutes,&stop.seconds);
        display(start,stop,&sum,&dif);
        printf("\nHours:%d\nMinutes:%d\nSeconds:%d\n",sum.hour,sum.minutes,sum.second);
        printf("\nHours:%d\nMinutes:%d\nSeconds:%d",dif.hour,dif.minutes,dif.seconds);
}
void display(struct TIME startt,struct TIME stopt,struct TIME *sumt,struct TIME *dift)
{
        int n=0;
        sumt->seconds=startt.seconds+stopt.seconds;
        if(sumt->seconds>60)
        {
                n=sumt->seconds/60;
                sumt->seconds=sumt->seconds%60;
```

```
        }
        sumt->minutes=sumt->minutes+startt.minutes+stopt.minutes;
        if(sumt->minutes>60)
        {
                n=sumt->minutes/60;
                sumt->minutes=sumt->minutes%60;
                sumt->hour=sumt->hour+n;
        }
        sumt->hour=sumt->hour+startt.hour+stopt.hour;
        dift->seconds=stopt.seconds-startt.seconds;
        if(dift->seconds<0)
        {
                dift->seconds=dift->seconds+60;
                dift->minutes=dift->minutes-1;
        }
        dift->minutes=dift->minutes+stopt.minutes-startt.minutes;
        if(dift->minutes<0)
        {
                dift->minutes=dift->minutes+60;
                dift->hour=dift->hour-1;
        }
        dift->hour=dift->hour+stopt.hour-startt.hour;
}
```

Example 8.1.10: Create a structure STUDENT containing name, symbol number, name of 6 subjects, mark of each subject and total mark as its members. Write a program that uses this structure and reads data for a student and gives the total marks as the output. [062/p/14]

```
#include<stdio.h>
void main()
{
        struct STUDENT
        {
                char name[50];
                int symbolno;
                char subjectname[6][50];
                int marks[6];
                int totalmarks;
        };
        struct STUDENT s;
        int i;
        s.totalmarks=0;
        printf("Enter name and symbol number:");
        gets(s.name);
        scanf("%d",&s.symbolno);
        for(i=0;i<6;i++)
        {
                fflush(stdin);
                printf("Name of subject %d",i+1);
                gets(s.subjectname[i]);
                printf("Enter marks of subject %d",i+1);
                scanf("%d",&s.marks[i]);
                s.totalmarks=s.totalmarks+s.marks[i];
        }
        printf("\nTotal Marks=%d",s.totalmarks);
}
```

Example 8.1.11: Create a structure named marks that has subject and mark as its members. Create another structure named student that has name, roll, marks and remarks as members. Assume appropriate type and size of members. Include first structure as a member for the second structure and read data for 10 students. The modified structure must have provision to store marks of three subjects for each students.

```
#include<stdio.h>
struct marks
{
        char subject[50];
        float mark;
};
struct student
{
        char name[50];
        int roll;
        char remarks[10];
        struct marks m[3];
};
void main(void)
{
```

```
            struct student s[10];int i,j;
            for(j=0;j<10;j++)
            {
                    printf("Enter name:");
                    scanf("%s",s[j].name);
                    printf("Enter roll:");
                    scanf("%d",&s[j].roll);
                    for(i=0;i<3;i++)
                    {
                            printf("Subject name:");
                            scanf("%s",s[j].m[i].subject);
                            printf("Marks:");
                            scanf("%f",&s[j].m[i].mark);
                    }
                    printf("Remarks:");
                    scanf("%s",s[j].remarks);
            }
            for(j=0;j<10;j++)
            {
                    printf("\nName:%s\nRoll:%d",s[j].name,s[j].roll);
                    for(i=0;i<3;i++)
                    {
                            printf("\n%s:",s[j].m[i].subject);
                            printf("%f",s[j].m[i].mark);
                    }
                    printf("\nRemarks:");
                    printf("%s",s[j].remarks);
            }
}
void linkfloat()
{
            float a=0,*b;
            b=&a;
            a=*b;
}
```

Example 8.1.12: Create a structure containing real and imaginary as its member. Write a program that uses this structure to input two complex numbers, pass them to a function to multiply them and return the result to main and display it.

```
#include<stdio.h>
struct complex
{
            int real;
            int imag;
};
void main()
{           struct complex num1,num2,result;
            struct complex product(struct complex,struct complex);  /* Local function declaration */
            printf("Enter read and imaginary part of num1:");
            scanf("%d%d",&num1.real,&num1.imag);
            printf("Enter read and imaginary part of num1:");
            scanf("%d%d",&num2.real,&num2.imag);
            result=product(num1,num2);
            printf("%d+i%d",result.real,result.imag);
}
struct complex product(struct complex num1,struct complex num2)
{
     struct complex p;
     p.real=num1.real*num2.real-num1.imag*num2.imag;
     p.imag=num1.real*num2.imag+num1.imag*num2.real;
     return p;
}
```

Output

```
Enter read and imaginary part of num1:2  4
Enter read and imaginary part of num1:6  8
-20+i40
```

8.1.11 Self-referential structures

- Structure that contains a pointer to a structure of the same type
- Can be linked together to form useful data structures such as lists, queues, stacks and trees
- Terminated with a NULL pointer.

Example:

```
struct node
{
            int data;
```

```
        struct node *nextPtr;
};
```

nextPtr

- Points to an object of type node
- Referred to as a link
- Ties one node to another node

The following figure shows a link list of self referential structures.

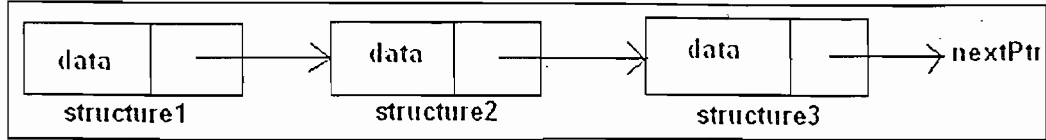


Fig.8.1.2 A list of self referential structures

8.2 Unions

Both structures and unions are used to group a number of different variables together. Union is another way of creating user defined data types. Syntactically both structures and unions are exactly same. Then what is the difference between structure and union? The main difference between is in storage. In structures, each member has it own storage location but all members of a union use the same memory location. Due to sharing of a common memory location, all the members of a union can not be accessed at any time. Only one member can be accessed at a time. The member to be accessed should be tracked by the programmer. The main objective of using union is to save memory space. Like structures, a union can be declared using the keyword **union**. For more clarification of the differences between structure and union, let us see definition and memory allocation of a structure and an union.

8.2.1 Definition

Structure	Union

struct student
{
    char name[30];
    int roll;
    float marks;
}
	union student
	{
	    char name[30];
	    int roll;
	    float marks;
	}

8.2.2 Memory allocation

Structure	30 bytes for name	2 bytes for roll	4 bytes for marks
Union	30 bytes for name		
Discussion	In case of structure memory is reserved for all the members separately but in case of union memory is reserved equal to the size of the largest member. Here, name is the largest member so only 30 bytes is reserved for the whole union.		

Example 8.2.1: This example illustrates the main difference between structures and unions.

```
union student
{       char name[30];
        int roll;
        float marks;
};
void main()
{
        union student s;
        printf("Size of union s is %d",sizeof(s));
        printf("\nEnter name:");
        scanf("%s",s.name);
        printf("\nName:%s",s.name);
        printf("\nEnter roll:");
        scanf("%d",&s.roll);
        printf("\nName: %s",s.name);
        printf("\nRoll: %d",s.roll);
        printf("\nEnter marks:");
        scanf("%f",&s.marks);
        printf("\nName: %s\nRoll: %d\nMarks: %f",s.name,s.roll,s.marks);
}
```

Chapter 9
File Input/output

9.1 Introduction

Sometimes, we need information be written to or read from an auxiliary memory devices. Such information can be stored on the memory devices in the form of a data file. Therefore a data file allows us to store information permanently. Which can be accessed to use or modify whenever necessary. Therefore, **A file is a place on the disk where a group of related data is stored**. The programs that we executed so far accepts the input data from the keyboard at the time of execution and writes output to the Video Display Unit. This type of I/O is called console I/O. For those operations, we have been using printf(), scanf(), getch(), getche(), getchar(), gets(), puts() etc. functions. Console I/O works fine as long as the amount of data is small. But, in many real life problems involves a large volume of data. In such situations, the console oriented I/O operations have two main problems:

- It becomes very inconvenient and time consuming to handle the large volume of data through the terminal.
- The entire data is lost when either the program is terminated or the computer system is turned off.

We need to create and use files to overcome these difficulties.

There are two types of data files: high level(stream oriented,standard) and Low level(system oriented). In this chapter, we will study only high-level files. The overall scheme of standard I/O and file I/O is illustrated in figure 9.1.

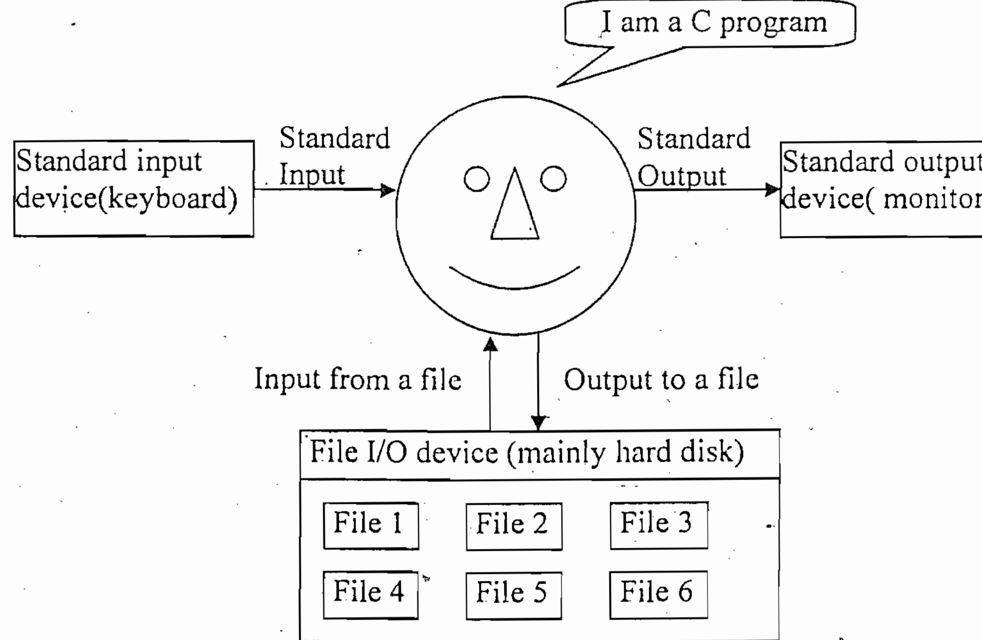


Fig. 9.1 Diagrammatic view of standard I/O and file I/O

9.2 Types of high level data files

High levels files are also subdivided into two categories:

- Text files
- Binary files

9.2.1 Text files

A text file is a human-readable sequence of characters and the words they form that can be encoded into computer-readable formats such as ASCII. A text contains only textual characters with no special formatting such as underlining or displaying characters in boldface or different font. There is no graphical information, sound or video files. A text file known as an ASCII file and can be read by any word processor. Text file stores information in consecutive characters. These characters can be interpreted as individual data items or as a component of strings or numbers. A good example of text file is any C program file, for example say **first.c**.

File operations

The main file operations are listed below and discussed in a little depth in the subsequent text. These operations are for both text mode and binary mode.

- Creating a new file
- Opening an existing file
- Writing information to a file
- Reading information from a file.
- Moving to a specific location in a file
- Closing a file

Let us start the discussion of these operations from example 9.1.

Example 9.1: This example first writes the data to the file named mydata.txt and read them back.

```
#include<stdio.h>
void main()
{
        FILE *fp;
        char ch;
        fp=fopen("D:\\kec\\mydata.txt","w");
```

Output and discussion

Size of union s is 30	Size of the union is found by sizeof operator.
Enter name: kamala Name: kamala	The value of union member name is read and displayed.
Enter roll: 501 Nmae: ð□mala Roll: 501	Now, value of structure member roll is read and name and roll both printed. But only Roll is printed correctly because memory space is same for all members. Therefore, value of name is overwritten by value of roll.
Enter marks:70.78 Name: \□□Bla Roll: -28836 Marks: 70.779999	In this case, only the value of marks is printed correctly. This discussion concludes that at any instant only one of the union members has meaningful value. Only that member which is the last written, can be used for computation. At this time, all other members have garbage value. But in case of structure all members can be accessed independently at any time.

Exercise 8

1. What is a structure? When do we use structures? [058/c/5-a]
2. How do we define structures?
3. What is the difference between normal variable declaration and structure variable declaration?
4. What are the different ways of accessing members of a structure?
5. What do we understand by the structure pointer operator? Explain with example. [062/p/13]
6. Differentiate between an array as a structure member and a structure as an array member with suitable examples.
7. Explain about nested structure with examples. [2061/ /7-a]
8. Is there any difference between array and structure? Explain with examples. [061/p/7-a]
9. What are different ways of passing a structure to a function? Explain with examples.
10. Define a union with an example.
11. What are the main differences between structures and unions?
12. What do we mean by user-defined data. Write different ways of creating user-defined data with examples. [062/b/7-a]
13. Write a program to read and display the Name, Address, Telephone number and Salary of an employee using structure. [057/c/7-b, 058/p/7-b]
14. Write a program to create a structure containing members to store name, roll number and mark of one subject of a student. Test your structure by reading and displaying the data for ten students. [058/c/5-b, 059/p/2-b (similar)]
15. Write a computer program to read and display the Name, Address, Account number and balance amount of a person using structure. [057/c/7-b, 058/p/7-b (similar)]
16. Create a structure containing processor, memory, HDD, monitor, casing as members. Use this structure in a program to read data (price) for five different computers and display the data on the screen. [059/c/6-b, 2063/b/11]
17. Write a program that contains a structure to hold code, name, price and stock of a product in a supermarket. Assume suitable data types. Use the structure in a program to input the 50 data of such products and display them. [059/p/2-b]
18. Write a program to create a structure student having members name, roll number and address. Member address have city and ward number as its member. Create an array of objects of type student. Read the value of corresponding elements in main function and pass the array to the function display () and display the information related to each member of the structure array.

```
        if(fp==NULL)
        {
                fprintf(stderr,"File mydata.txt can not be opened");
                exit();
        }
        fprintf(fp,"This is my first file I/O program.");
        fclose(fp);
        fp=fopen("D:\\kec\\mydata.txt","r");
        if(fp==NULL)
        {
                fprintf(stderr,"File mydata.txt can not be opened");
                exit();
        }
        while(1)
        {
                ch=fgetc(fp);
                if(ch==EOF)
                        break;
                else
                {
                        putchar(ch);
                        /*      printf("%c",ch);
                                fprintf(stdout,"%c",ch);
                                fputc(ch,stdout);       */
                }
        }
}
```

Let us understand example 9.1 – While working with a file, at first, we need to declare a pointer of type FILE. FILE is a structure. Which is discussed in subtopic 9.6. This program declares a pointer fp, which can hold the FILE pointer. We need to open the file before doing any operations to the file. The **fopen** function takes two strings as arguments. The **first argument** is the file name with its location (path). If we omit the path, the function **fopen** searchers the requested file in the bin. While opening a file, we should specify the purpose of opening the file. There are different types of purposes which are called opening modes (see 9.2.1.6). We need to specify the mode of the file being opened as the second argument of the function **fopen**. If **fopen** function opens the file appropriately, it returns the pointer of the file to the file pointer fp. In the above example, **fopen** opens the file mydata.txt in read mode. If **fopen** can not open the requested file, it will returns the NULL pointer which is assigned to fp. We need to check whether the required file opened successfully or not. This task is done by checking the value of fp. If fp is NULL, it is required to put some information on the screen. Which is done in the above example by printing, " File data.txt can not be opened" on the standard error device (monitor). The function **exit** is a special function that terminates the program immediately. **exit(0)** mean that we wish to indicate that the program terminated successfully whereas a nonzero value means that the program is terminating due to an error condition.

After opening the file, a line of text " **This is my first file I/O program.**" is written to the file named myfile.txt. Here, function **fprintf** is used to write to the file. Which is similar to function printf. The function **fclose** is used to close the file. The file is again opened in the read mode. In this example, file is read character by character. Function **fgetc** is used to read characters from the file having pointer fp. fp is used as an argument of function fgetc. Then it is checked that whether the read character is other or EOF(end of file). If it is EOF, the indefinite while loop is broken using break statement. Otherwise, this read character is priented on the screen using function **putchar.** Instead of putchar, we can use other equivalent syntax, which are listed below **putchar** function. After finishing read operation, file is again closed. As we know, most of the above listed file operations are included in this discussion. All these operations are discussed below biefly.

9.2.1.1 **Creating a new file**: If there is no already created file in the disk, we need to create a new file to write information. File opening modes w, w+, a, a+ can be used to create a new file. In example 9.1 mydata.txt is created initially.

9.2.1.2 **Opening an existing file:** Before reading data (information) from an existing file we need to open in read mode. In fact, fopen performs following important tasks when we open the file in "r" mode.
Firstly, it searches on the disk to the specified path to open the file.
If it finds the file, it loads the file from the disk into to the buffer and it sets up character pointer that points the first character of the buffer.

What is a buffer?
A section of Random Access Memory (RAM) reserved for temporary storage of data waiting to be directed to a device.
Why buffer is needed?
Case study- let us consider, our program accesses a disk to read/write one character at a time. The system must spend some time to position the read/ write head on the correct tract and again waits for the correct sector of the track to read/write operation. The motor that rotates the disks starts from the stand still for every read/write operation. Form all above discussion, we know that it is required to spend a long duration for each read/write operation. When we write several characters in a short interval, there will be the probability of loss of any character. To solve such kinds of problem, the buffer is required. The buffer acts like a reservoir, capturing the data and then letting it out at speeds that the receiving device can handle without overflowing. When a program writes data to a file, it is stored in the buffer till

the buffer has space. When the buffer is full, its content is written to the disk all at once and subsequent writes continues on to the buffer. Similarly, when reading the data form a file on the disk, at first, a block of data is read and placed in the buffer. The size of each block is equal to the size of block except the last block. Then, the program reads the data from the buffer. When a program tries to read data, which is not in buffer, then next block containing the requested data should be brought to the buffer by replacing the previous block to the disk. In conclusion, A buffer is used to aid communication between two devices with very different processing speeds. (e.g. RAM and hard disk CPU and the printer). If we try to open an existing file in **w** mode, all the content will be destroyed. If the file opening fails due to any reason (see subtopic 9.4 for reasons), fopen returns a value NULL. Then the system must be handled properly. The way of handling such situation is illustrated in **example 9.1and 9.10.** In many of the following example, it is assumed that files are opened successfully without any troubles. While writing **program,** we must handle the situation properly.

9.2.1.3 Writing to a file

Before writing something to a file, it must be opened in w mode. Before writing something to a file on the disk, it should be written to the buffer. If the buffer become full, then the content of buffer is sent to the corresponding file pointed by the FILE pointer. If the buffer is not full, but file is closed, in such situation, the content of file is also written to the file on disk. We can write using function fputc or macro putc(see chapter 10). In each file reading function, there should be a file pointer as an argument. Other part is similar to the functions that are used for reading from standard input device(key board). In the above example, we have written a line of text using **fprintf**. In **fprintf,** FILE pointer fp is used as the first argument. Other part is similar to the function printf.

9.2.1.4 Reading information from a file

To read information from a file, the file must be opened in read mode. As we discussed, to read anything from a file, it must be brought to the buffer and pointer must be set to the first character. In example 9.1, we have used function fgetc to read the content of memory pointed by the file pointer fp. It returns the read character to the character variable ch. After reading the pointed character, fgetc advances the pointer to point to the next character. We have used the function getc in the indefinite while loop. There has to be a way to break the idefinite while loop. A special character having ASCII value 26, indicates the end of file. When a file is created, the character is inserted beyond the last character in the file. If **fgetc** encounters the special character while reading a file, it returns macro EOF instead of returning that character. EOF has been defined in stdio.h. Instead of using function **fgetc,** we can use macro **getc** to read a character.

9.2.1.5 Moving to a specific location in a file

This sub topic is discussed in example 9.10 while discussing function **fseek.**

9.2.1.6 Closing a file

When we have finished reading, writing or appending operation, we need to close the file. Function **fclose** does this job. That is illustrated in example 9.1. After closing a file, we cannot do any operations on the file. When we close a file using function **fclose,** three operations would be performed.

- The characters in the buffer would be written to the file on the disk.
- A character with ASCII value 26 would get written at the end of the file.
- The buffer associated with the file is removed from the memory.

If the buffer is filled before closing the file, the content of the buffer must be written to the disk at the moment it becomes full.

9.2.1.6 File Opening Modes

While opening file in any mode, function **fopen** searches the file as the first task. Other tasks are discussed individually in the following subtopics. Where different types of file opening modes are discussed.

r, r+ : If the searched file is found, function fopen() loades it into memory and sets up a pointer which points to the first characters in it. If searched file is not found, function fopen() returns NULL.

Operations possible

r : Reading from the file only.

r+: Reading existing content, writing new contents and modifying existing contents of the file.

w, w+ : If the searched file is found, its content is overwritten. If the file is not found, a new file is created. If function fopen() unable to open the file, it returns NULL.

Operations possible

w: writing to the file only

w+: writing new contents, reading them back and modifying the existing content of the file.

a, a+ : If the searched file is opened successfully, function fopen() loads the file in to memory and sets up a pointer which points to the last character in it. If the searched does not exists, a new file is created. fopen() returns NULL, if unable to open file.

Operations possible

a : adding new contents at the end of file.

a+ : appending new content to the end of file, reading existing content from the file. But, can not modify existing contents.
In text mode, we can add t to the mode string as at, at+, wt, wt+, rt, rt+. Text mode is default mode so t can be omitted.

9.2.1.7 Different types of file I/O

In most of the following example, it is assumed that all the files are opened in the required mode successfully. [while writing progrm do not assume.]

9.2.1.7.1 Character I/O

Example 9.2: This example illustrates the following concepts:

- Reading character from keyboard using macro **getchar** (we can also use function **getche**)
- Writing them to the file(first.txt) pointed by the FILE pointer fp using function **fputc**.
- Closing the file, before opening it in different mode.
- Use of **fgetc** function to read characters from the file.
- Displaying the read character on the screen using macro **putchar**.

```
#include<stdio.h>
void main(void)
{
        FILE *fp;
        char ch;
        fp=fopen("first.txt","w");
        printf("Enter characters.");
        do
        {       ch=getchar();
                fputc(ch,fp);
        }while(ch!='\n');
        fclose(fp);
        fp=fopen("first.txt","r");
        while(1)
        {
                ch=fgetc(fp);
                if(ch==EOF)
                        break;
                else
                        putchar(ch);
        }
}
```

Example 9.3: A file copy program. This program writes something in the source file, copies the content of the source file to the destination file and displays the content in the destination on the screen.

```
#include<stdio.h>
void main()
{
        char sfname[50], dfname[50], ch; /* Declaration of string variables to store file name.s */
        FILE *sfpointer,*dfpointer; /* Declaration of source and destination FILE pointers.*/
        printf("Enter source file name:");
        gets(sfname);
        printf("Enter destination file name:");
        gets(dfname);
        sfpointer=fopen(sfname,"w"); /*Opening source file in w mode .*/
        fprintf(sfpointer," I am writhing something to source file before copying.");
        fclose(sfpointer); /* Closing source file after writing something in it. */
        sfpointer=fopen(sfname,"r"); /* Opening source file in r mode. */
        dfpointer=fopen(dfname,"w"); /* Opening destination file in w mode. */
        while(1)
        {
                ch=fgetc(sfpointer);
                if(ch==EOF)
                        break;
                else
                        fputc(ch,dfpointer); /* Writing character to the fiile having pointer dfpointer */
        }
        fclose(dfpointer);
        dfpointer=fopen(dfname,"r");
        while(1)
        {
                ch=fgetc(dfpointer);
                if(ch==EOF)
                        break;
                else
                        putch(ch); /* Displaying the copied content of destination file on the screen. */
        }
        fclose(sfpointer);
        fclose(dfpointer);
}
```

Sometimes, it is more convenient to read from or write to characters in the form of strings of characters. We have already used function **gets** and **puts** for console I/O. Similarly, we have functions **fgets** and **fputs** for file I/O

Example 9.4: This example illustrates the concept of string I/O in a file.

```
#include<stdio.h>
void main(void)
{
        FILE *fp;
        char str[100];
        fp=fopen("second.txt","w");
        printf("Enter some strings:");
        while(1)
        {
                        gets(str);
                        if(strlen(str)>0)
                        {
                                fputs(str,fp);
                                fputs("\n",fp);
                        }
                        else
                        break;
        }
        fclose(fp);
        fp=fopen("second.txt","r");
        while(fgets(str,99,fp)!=NULL)
        {
            puts(str);
        }
        fclose(fp);
}
```

In example 9.4, function gets read the string, assigns it to the character array str, fputs function writes it to the file having FILE pointer fp. This job is being done in the while loop until the user enters a string of length zero i.e., user hits the enter key. Then, the file is closed and again it is opened in read mode. Function **fgets** reads the strings from the file having FILE pointer fp and assigns to the string str. Which is displayed back to the VDU(video display unit) using function **puts**. This job is continued till the **fgets** reads the NULL character. If all the lines are read and any more attempt to read another line, fgets returns NULL. Note that, function fgets takes two arguments: one is the pointer to structure FILE the and another is the string variable name where the string to be written to the file is stored. Similarly, function **fgets** takes three arguments. First is the address(name of the array) where the string is stored. Second is the maximum length of the string. The function fgets can not read more characters then the size of array so the second argument prevents the overflow of the array. Third is the pointer to the tructure FILE.

9.2.1.7.3 Formatted I/O(General I/O)

We have already known function printf for formatted out put and scanf for formatted input. Similarly, functions **fprintf** and **fscanf** are used for formatted I/O in a file. All the concept of printf and scanf is similar to functions fprintf and fscanf except first argument. The first argument of fprintf and fscanf is a pointer to the structure FILE.

Example 9.5: Illustration of Formatted I/O in a file.

```
#include<stdio.h>
void main(void)
{
     FILE *fp;
     char name[100],next[3];
     int roll;
     float marks;
     fp=fopen("third.txt","w+");
     do
     {
            printf("Enter name,roll and marks:");
            scanf("%s%d%f",name,&roll,&marks);
            fprintf(fp,"%s %d %f",name,roll,marks);
            printf("If you do not have next data set, enter \"no\" otherwise press any key and enter. ");
            scanf("%s",next);
     }while(strcmp(next,"no")! =0)
     rewind(fp);
     while(fscanf(fp,"%s%d%f",name,&roll,&marks)!=EOF)
     {
            printf("Name: %s\n Roll: %d\n Marks: %f\n",name,roll,marks);
     }
     fclose(fp);
}
```

This program declares variables of different data types e.g., character, integer and float. Opens the file in write mode, goes into the indefinite while loop, asks the user to enter name roll and marks of a student, at first these are assigned to their respective variables, then these data are written to the file using function fprintf, asks the user to enter string "no" if

he/she is satisfied, looping continues until the user enters "no". Similarly, Using another while loop all the members are read from the file. They should be read in the same format as in they were written. After scanning the data items from the file, they are assigned to the corresponding variables specified in the function **fscanf**. At last, they are displayed on the VDU using function printf. Note that one new concept is used here. The file is opened in w+ mode, Where both writing and reading can be done. But before reading we should bring the pointer to the required position. Here, the pointer fp is repositioned to the starting character using the function rewind. The reading loop runs until fscanf returns EOF. Then, all the content is read form the file and displayed on the screen.

Illustration of r+

Example 9.6 This example is the modification of example 9.5. It first reads the content of the file third.txt and displays on the screen. Then, it reads name, roll and marks of students until the user enters 'n' after reading data, write it to the file than display the content of the file.

```
#include<stdio.h>
void main(void)
{
        FILE *fp;
        char name[100], ch;
        int roll;
        float marks;
        fp=fopen("third.txt","r+");
        if(fp==NULL)
                exit();
        while(fscanf(fp,"%s%d%f",name,&roll,&marks)!=EOF)
        {
                printf("Name: %s\n Roll: %d\n Marks: %f\n",name,roll,marks);
        }
        while(1)
        {
                printf("Enter name,roll and marks:");
                scanf("%s%d%f",name,&roll,&marks);
                fprintf(fp,"%s  %d  %f ",name,roll,marks);
                printf("If you do not have next data set, enter n ");
                scanf("%c",ch);
                if(ch== 'n')
                break;
        }
        rewind(fp);
        while(fscanf(fp,"%s%d%f",name,&roll,&marks)!=EOF)
        {
                printf("Name: %s\n Roll: %d\n Marks: %f\n",name,roll,marks);
        }
        fclose(fp);
}
```

Similarly, we can do in a+ mode.

9.2.1.7.4 Record I/O in files

So far we discussed about character I/O, string I/O and formatted I/O (General I/O). In this chapter we are going to discuss about structure I/O in files. Structures can be written to or read from the file as in the above example. Which is illustrated in example 9.7.

Example 9.7 : Illustration of w+, for loop, fscanf and accessing structure members individually.

```
#include<stdio.h>
void main(void)
{
        FILE *f;
        int i;
        struct student
        {
                char name[30];
                int marks;
        };
        struct student s;
        if((f=fopen("student.txt","w+"))= =NULL)
                exit();
        for(i=0;i<20;i++)
        {
                printf("Enter name and marks:");
                fscanf(stdin,"%s%d",s.name,&s.marks);
                fprintf(f,"%s %d",s.name,s.marks);
        }
        rewind(f);
        while(fscanf(f,"%s%d",s.name,&s.marks)!=EOF)
        {
                fprintf(stdout,"Name : %s\n Marks : %d\n",s.name,s.marks);
        }
```

```
/*      for(i=0;i<20;i++)
        {
                fscanf(f,"%s%d",s.name,&s.marks);
                fprintf(stdout,"Name : %s\n Marks : %d\n",s.name,s.marks);
        }       */
    fclose(f);
}
```

The working process of the above example is similar to the example in formatted I/O except the concept of structure and using data item as structure member. Here, number of students is limited to 20. Therefore, for loop is used to read form keyboard and writing to the file. But, to read from the file, we can use while loop as in the previous examples or for loop. The syntax of for loop is shown as comment.

9.2.2 Binary Files

Binary files contain more than plain text e.g., sound, image, graphic etc. In contrast to ASCII files, which contain only characters (plain text), binary files contain additional code information. A binary file is made up of machine-readable symbols that represent 1's and 0's. The binary file content must be interpreted by a program that understands in advance exactly how it is formatted. These files organize data in to blocks containing contagious bytes of information. These blocks represent more complex data structures e.g., arrays and structures. A good example of binary file is the executed C program file. For example, **first.exe** is a binary file. A very easy way to find out whether a file is a text or a binary file is to open that file in Turbo C/C++. If we can read the content of the file, then, it is a text file else binary file. Other examples of binary files are sound files, graphics files etc.

Opening Modes : The file opening modes in binary files are similar to text mode. Which are rb, rb+, wb, wb+, ab and ab+. The nature of each mode is similar to the mode of text file. Character b is added to each mode to indicate the binary mode.

9.2.2.1 Differences between text and binary file

There are three main differences between text and binary mode files. Which are:

a. **Handling of new lines**
b. **Representation of end of file**
c. **Storage of numbers**

Which are discussed in detail in the following section.

a. Handling of new lines

In text mode, a new line character is converted into carriage return-linefeed combination before writing it to the disk. While reading back from the disk, the carriage return-linefeed combination is converted back into a new line. If a file is opened in binary mode, these conversions are not required.

b. Representation of end of ~~line~~ file

In text mode, a character having ASCII value 26 is inserted at the end(after the last character) of the file to mark the end of file. The read function would return EOF signal, if this character is encountered while reading the file. Which indicates the end of file. There is no such special character at the end of binary file to mark the end of file. The binary mode files keep track of the end of file from the number of characters present in the directory entry of the file.

Question

Suppose one file stores numbers in binary mode, a umber 26 is also stored in the file. But , we need to read the numbers in the text mode.
What would happen when the number 26 is encountered before reaching to the end of file?

Answer

Reading would be terminated permanently at the point where 26 is encountered because ASCII value 26 means end of file.

From the above question/answer we conclude that these two modes are not compatible. So, any file written in text mode must be read back in text mode and written in binary mode must be read in binary mode.

c. Storage of numbers

If someone asks us, what is the size of integer in memory? We can easily say 2 bytes. But while storing number in text mode, each digit is stored as a character, which occupies one byte. Which means numbers are stored as string of character. The number 32235 occupies two bytes in memory but when it is written to disk using function fprintf, it occupies 5 bytes. Similarly, 12345.67890 occupies 4 bytes in memory but 11 bytes in disk.

From the above discussion, we conclude that if large amount of numerical data is to be stored in a disk file in text mode, there might be insufficient space on the disk. If the file is opened in binary mode then the above problem will be solved. In binary mode, we use functions **fread** and **fwrite** which stores the numbers in the binary format. It means each number would occupy the same numbers of bytes on both memory and disk.

9.2.2.2 Record I/O in binary mode

Record I/O in example 9.7 has following problems:

a. The file was in text mode. The number marks would occupy more number of bytes because each number is stored as a character string.

b. If we add more member to the structure student e.g., student's roll no, section, age, address, phone number etc, it will be very uncomfortable to write them to file using function **fprintf** and reading them back using function **fscanf**.

There are functions **fread** and **fwrite** for reading from and writing to structure (record) in a file on the disk. Which is illustrated in example 9.8.

Exampe 9.8 : Illustration of fread / fwrite functions

```
#include<stdio.h>
struct employee
{
        char name[100];
        int salary;
        float age;
};
void main()
{
        struct employee emp, e;
        FILE *filepointer;
        char filename[30];
        int decision=1;
        printf("Enter file name:");
        scanf("%s",filename);
        if((filepointer=fopen(filename,"wb"))= =NULL)
                exit();
        while(decision)
        {
                printf("Enter name, salary and age of employees:");
                scanf("%s%d%f",emp.name,&emp.salary,&emp.age);
                fwrite(&emp,sizeof(emp),1,filepointer);
                printf("Enter \n0 to exit:\n1 to continue to read next data set:");
                scanf("%d",&decision);
        }
        fclose(filepointer);
        if((filepointer=fopen(filename,"rb"))= =NULL)
                exit();
        while(fread(&e,sizeof(e),1,filepointer)= =1)
        printf("Name : %s\nSalary : %d\nAge : %f\n",e.name,e.salary,e.age);
        fclose(filepointer);
}
void linkfloat()
{
        float a,*b;
        b=&a;
        *b=0;
}
```

This example reads the structure members from the standard input device. Writes them to the file using function **fwrite**. It takes four arguments: the **first** is the address of the structure to be written to the disk, the **second** is the size of the structure in bytes, the **third** is the number of such structures that we want to write at a time and the **fourth** is the pointer to the file where we want to write. Then, the written contain is read back using function **fread**. Which takes the similar four arguments as in **fwrite**: address of structure, size, number of structures to be read at a time and file pointer. It returns the number of records it read. In this example, it returns 1. After reaching at the end of file, fread can not read anything and it returns 0. Which indicates to stop the reading and exiting from the while loop. Function fread causes the data read from the disk and to be placed in the structure variable **e**. We can also use the structure variable **emp**. Note that, in this example, file is opened in binary mode. After reading from file to memory, the structure members are displayed on the screen using function printf. Do you know why function **linkfloat** is used?

Example 9. 9: Write a program to read name, rollno and marks obtained by students in five subjects and store the result in a file until the user is not satisfied. The program should have another portion to read all information. Your program should ask the user to read from or write to the file and call the specific functions.[062/b/8] [Note : call write_to_file() before calling read_from_file()]

```
#include<stdio.h>
void write_to_file();
void read_from_file();
struct student
{
        char name[50];
        int rollno;
        float marks[5];
};
void main()
{       int i;
        printf("enter \n1 for writing to file.\n2 for reading from file");
```

```
        switch(i)
        {       case 1:
                        write_to_file();
                        break;
                        case 2:
                        read_from_file();
                        break;
                        default:
                        printf("Enter 1 or 2");
                        break;
        }
}
void write_to_file()
{       FILE *f; struct student s;int i;char next;
        f=fopen("c:\\tc\\bin\xyz\\abcd1.dat","wb");
        do
        {  printf("\nfEnter Name:");
           scanf("%s",s.name);
           printf("\nEnter rollno:");
           scanf("%d",&s.rollno);
           for(i=0;i<5;i++)
           {
                        printf("Enter marks of subject[%d]",i+1);
                        scanf("%f",&s.marks[i]);
           }
           fwrite(&s,sizeof(s),1,f);
           printf("PRESS y for next item \nelse any key");
           next=getche();
        }while(next= ='y');
        fclose(f);
}
void read_from_file()
{
        FILE *fp;struct student s;
        int i;
        fp=fopen("c:\\tc\\bin\\xyz\\abcd1.dat","rb");
        while(fread(&s,sizeof(s),1,fp)= =1)
        {
                printf("Name=%s\nrollno=%d\n",s.name,s.rollno);
                for(i=0;i<5;i++)
                {
                        printf("marks %d =%f\n",i+1,s.marks[i]);
                }
        }
        fclose(fp);
}
```

9.3 Database Management

A database is a collection of data that is organized so that its contents can easily be accessed, managed and updated. Software package that allows us to use a computer to create a database, add, change, search, retrieve etc. is called a database management system.

Case study : Suppose, there are 40 students in your class. You need a software to manage the information related to each student. The software should have the following facilities:

a. Creating a database to store information.
b. Adding new records to the database.
c. Displaying a list of all records in the database.
d. Modifying the existing records in the database.
e. Deleting the unwanted records from the database.
f. Searching the required records in the database.

Then what will you do?

All the above requirements are implemented in example 9.10. Study and understand.

Example 9.10:

```
#include<stdio.h>
#include<conio.h>
struct student
{
        char name[100];
        int roll;
        float marks;
} s;
void Addrecords();
void Listrecords();
void Modifyrecords();
void Deleterecords();
```

```
void Searchrecords();
char filename[30];
void main(void)
{
        char choice;
        printf("Enter file name:");
        scanf("%s",filename);
        clrscr();
        while(1)
        {
                printf("\nEnter \n A or a : For adding new records.\n L or l : For displaying a list of all the records");
                printf("\n M or m : For modifying a record \n D or d : For deleting a record.\n S or s : For searching a record.\n E or e : For exit.\n ");
                fflush(stdin);
                choice=getche();
                switch(choice)
                {
                        case 'A':
                        case 'a':
                        Addrecords();
                        break;
                        case 'L':
                        case 'l':
                        Listrecords();
                        break;
                        case 'M':
                        case 'm':
                        Modifyrecords();
                        break;
                        case 'D':
                        case 'd':
                        Deleterecords();
                        break;
                        case 'S':
                        case 's':
                        Searchrecords();
                        break;
                        case 'E':
                        case 'e':
                        exit();
                        default:
                        printf("Enter A,a,L,l,M,m,D,d,E or e");
                }
        }
}
void Addrecords()
{
        FILE *f;
        char test;
        if((f=fopen(filename,"ab+"))= =NULL)
        {
                if((f=fopen(filename,"wb+"))= =NULL)
                {
                        printf("Can not open the file.");
                        getch();
                        exit();
                }
        }
        while(1)
        {       printf("\nEnter name, roll, marks:");
                scanf("%[^\n]%d%f",s.name,&s.roll,&s.marks);
                fwrite(&s,sizeof(s),1,f);
                fflush(stdin);
                printf("Enter Esc key to exit:");
                test=getche();
                if(test= =27) /* 27 is the ASCII value of Esc key. */
                break;
        }
        fclose(f);
}
void Listrecords()
{
        FILE *f;
        if((f=fopen(filename,"rb"))= =NULL)
        exit();
        clrscr();
        printf("\nName                    Roll        Marks\n");
```

```
        while(fread(&s,sizeof(s),1,f)==1)
        {
                printf("\n%-40s%-10d%10.2f",s.name,s.roll,s.marks);
        }
        fclose(f);
        getch();
}
void Modifyrecords()
{
        FILE *f;char name[100];long int size=sizeof(s);
        if((f=fopen(filename,"rb+"))==NULL)
        exit();
        printf("\nEnter name of student to modify:");
        scanf("%[^\n]",name);
        fflush(stdin);
        while(fread(&s,sizeof(s),1,f)==1)
        {       if(strcmp(s.name,name)==0)
                {
                        printf("\nEnter name, roll, marks:");
                        scanf("%[^\n]%d%f",s.name,&s.roll,&s.marks);
                        fseek(f,-size,SEEK_CUR);
                        fwrite(&s,sizeof(s),1,f);
                        break;
                }
        }
        fclose(f);
}
void Deleterecords()
{
        FILE *f,*temp;char name[100];
        if((f=fopen(filename,"rb"))==NULL)
        exit();
        if((temp=fopen("tempfile","wb"))==NULL)
        exit();
        printf("\nEnter name of student to Delete:");
        scanf("%[^\n]",name);
        while(fread(&s,sizeof(s),1,f)==1)
        {    if(strcmp(s.name,name)==0)
                continue;
                else
                fwrite(&s,sizeof(s),1,temp);
        }
        remove(filename);
        rename("tempfile",filename);
        fclose(f);
        fclose(temp);
}
void Searchrecords()
{
        FILE *f;char name[100];int flag=1;
        if((f=fopen(filename,"rb+"))==NULL)
        exit();
        printf("\nEnter name of student to search:");
        scanf("%[^\n]",name);
        while(fread(&s,sizeof(s),1,f)==1)
        {
                if(strcmp(s.name,name)==0)
                {
  printf("\nName : %s\nRoll : %d \nMarks : %f\n",s.name,s.roll,s.marks);
                        flag=0;
                        break;
                }
        }
        if(flag==1)
        printf("Record not found.\n");
        fclose(f);
}
```

In example 9.10,

- Structure **student** is defined globally and structure variable s is also declared globally.
- There are five different functions to do specific tasks.
- Main function requests the user to enter his/her choice.
- Switch statement is used to call the corresponding function according to user's choice.

Function Addrecords()

Tries to open the existing file in rb mode, if it is not found it opens new file in wb mode.

Reads the information form the user, assigns then in the corresponding fields of s.
Function **fwrite** writes the record as a single block to the file.
This process goes on until the user enters Esc key.
Function **fflush** - Flushes (empties) the buffers associated with the file.

Function Listrecords()

- Reads all the records and displays on the screen.

Function Searchrecords()

- Asks the user to input the name of the student to be searched and tries to search.
- If the record is found, it would be displayed on the screen.
- Searching is done by reading each record and comparing the given name with the read name.

Function Deleterecords()

- Asks the user to enter the name of student to delete his record.
- This function opens one temporary file.
- It reads records from the database (original file).
- Writes the read records to the temporary file.
- This process goes on until the record having the name of student to be deleted is found.
- If the searched record is found, it would not be written to the temporary file.
- Then all the remaining records are read from the original file and written to the temporary file.
- The Macro **remove** deletes the file specified by filename. It simply translates its call to a unlink. On success, it returns 0. On error, it returns –1. Here original file is deleted.
- Function **rename** changes the name of a file from oldname to newname. On success, it returns 0. On error, it returns –1. Here name of temporary file is renamed to the original file.

Function Modifyrecords()

- Searches the record to modify.
- Asks the user to enter the new data.
- Function **fseek** repositions the **pointer** back by the size of structure student form the current position. And overwrite the old record with the new record.

Function **fseek** has three arguments. The first is the FILE pointer, second is the number of bytes the pointer be moved from a particular position. The third argument could be SEEK_END, SEEK_CUR or SEEK_SET. These all act as the references from which the pointer should be repositioned. All these are macros defined in stdio.h. SEEK_END means move the pointer from the end of the file, SEEK_CUR means move the pointer from the current position and SEEK_SET means move the pointer with reference to the beginning of the file. We find the current position of the pointer by using **ftell** function. It returns the position of the pointer as a long int.
Above discussion of file I/O is in sequential manner. But we can access file randomly with the help of functions **ftell**, **fseek** and **rewind** available in stdio.h.

9.4 Error handling

In most of the examples, it is assumed that all the files are opened successfully. It is possible that an error may occur during I/O operation in a file. Therefore, **it is required to check** whether the required file is opened successfully or not in all the cases while opening a file. Most of the error occurring situations are:

- Trying to open a file with an invalid file name.
- Trying to write to a write protected disk or file.
- Trying to perform an operation on a file, when the file opened for another mode.
- Trying to use a file that has not been opened.
- Trying to read beyond the end-of-file mark.
- Trying to write to a filled disk.
- Trying to access a damaged disk.

If we could not handle such read/write errors properly, a program may terminate abnormally or may give incorrect output. We have two status-inquiry macros **feof** and **ferror** that can help us to detect I/O errors in the file.
feof : It can be used to test for an end of file condition. It takes FILE pointer as its only argument. It return non-zero integer when the end-of-file is reached otherwise it returns 0. If **filep** is a pointer to the file that has just been opened for reading, then the code segment

```
if(feof(filep))
printf("End of file.");
```

Would display the message " End of file." on reaching the end of file condition. The condition in if statement is always zero (means false) until the end-of-file is encountered. But at the end, the condition becomes non – zero (means false) and the body of if will be executed.
ferror: It reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has detected up to that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp)!=0)
printf("An error has occurred during processing");
```

would print the error message, if the reading is not successful.

If file can not be opened for some reasons, then the function fopen returns a NULL pointer. It can be used to check whether the file has been opened successfully or not opened. Which is illustrated in examples 9.1 and 9.10.

9.5 Standard I/O devices

As we know, to perform any I/O operations on a file we need to use function **fopen**, which sets up a pointer to refer to this file. MS-DOS also predefines pointers for five standard files. To access these standard file pointers, we need not use function fopen. These standard pointers are also called predefined streams. **A stream is a sequence of bytes.** These streams are automatically opened when a C program starts executing and are closed when the program terminates. The programmer doesn't need to take any special action to make these streams available. Table lists the standard streams and the devices they normally are connected with. All five of the standard streams are text-mode streams
Most operating systems predefine pointer for the first three streams.

Table 9.1 pointers to standard I/O devices

Name	Meaning	Purpose
Stdin	Standard input device [Key board]	opened for input
Stdout	Standard output device [VDU]	opened for output
stderr	Standard error output device [VDU]	opened for output
Stdaux	Standard auxiliary device [serial port]	opened for both input and output
Stdprn	Standard printer [Parallel printer]	opened for output

Till this point, to print a line " I am learning C", we have been using function printf in the following form.
printf("I am learning C");
In stead of using this, we can use the following syntax using function **fprintf** and file pointer stdout. Which pointes the standard output device[Monitor]
fprintf(stdout, "I am learning C");
Similarly, we can use fscanf(stdin, "%d",&a); to get value of variable a from keyboard.
If we want to print a string "Muskan", we can use the syntax fprint(stdprn," Muskan");
Similarly, we can use stderr to write something on the standard error device. The standard error device is also monitor (Video Display Unit). stdaux is used for reading or writing something to serial port (COM-1).

9.6 FILE structure

I/O functions read and write from the file streams. FILE stream structure is defined in the header file stdio.h. Which is shown below. It is not a good practice to user the members of the FILE structure for other purpose.

```
typedef struct
{
        int level;                        /* fill empty level of buffer*/
        unsigned flags;                   /* File status flags */
        char fd;                          /* File descriptor(handle) */
        unsigned char hold;               /* Ungetc char if no buffer */
        int bsize;                        /* Buffer size */
        unsigned char _FAR *buffer;       /* Data transfer buffer */
        unsigned char _FAR curp;          /* Current active pointer */
        unsigned istemp;                  /* Used for validity checking */
        short token;                      /* There is the FILE object */
}
```

9.7 Some Important examples

Example 9.11: Write a program that reads characters from keyboard and writes to a data file. The program should display the numbers of words entered and terminate when the user enters a dot. [059/p/7-b]

```
#include<stdio.h>
void main()
{
    char ch;
    int count=0;
    FILE *fp;
    fp=fopen("record.dat","w");
    printf("Enter a string\n");
    while(1)
    {
            ch=getche();
            if(ch=='.')
                    break;
            else
            {
                    fputc(ch,fp);
                    if(ch==' ')
                    count++;
            }
    }
    printf("\nNumber of words in the entered string is %d.",count+1);
    fclose(fp);
```

```
}
```

Example 9.12: Write a program to read name and roll number of 24 students from the user and store them in a file. If the file already contains data, your program should add new data at the end of the file. [058/c/7-b, similar concept to 059/c/7-b, similar concept to 060/p/7-b]

```
#include<stdio.h>
void main(void)
{
	char name[50],ch;
	int rollno;
	int i;
	FILE *fp;
	fp=fopen("students.txt","a");
	for(i=0;i<24;i++)
	{
		printf("\nEnter name:");
		scanf("%s",name);
		printf("Enter rollno:");
		scanf("%d",&rollno);
		fprintf(fp,"\nName:%s\nRoll no:%d\n",name,rollno);
	}
	fclose(fp);
	fp=fopen("students.txt","r");
	while(1)
	{
		ch=fgetc(fp);
		if(ch==EOF)
			break;
		else
			putchar(ch);
	}
	fclose(fp);
}
```

Example 9.13: Write a program to open a new file, read roll number, name, address and phone number of students until the user says "no" after reading the data, write it to the file then display the content of the file. [061/b/8, 2061/c/8-b]

```
#include<stdio.h>
#include<string.h>
void main(void)
{
	struct student
	{
		char name[80];
		char address[150];
		char phoneno[25];
		int rollno;
	} s;
	int i;char next[10];
	FILE *fp;
	fp=fopen("students.dat","wb");
	do
	{
		fflush(stdin);
		printf("\nEnter Name:");
		gets(s.name);
		fflush(stdin);
		printf("\nEnter Adress:");
		gets(s.address);
		fflush(stdin);
		printf("Enter phone no:");
		gets(s.phoneno);
		printf("Enter rollno:");
		scanf("%d",&s.rollno);
		fwrite(&s,sizeof(s),1,fp);
		printf("If you do not have next data set neter  no  \nelse press any key.");
		scanf("%s",next);
	}while(strcmp("no",next));
	fclose(fp);
	fp=fopen("students.dat","rb");
	while(fread(&s,sizeof(s),1,fp)==1)
	{
		printf("Name:%s\nAddress : %s\nPhone : %s\nRoll no : %d\n",s.name,s.address,s.phoneno,s.rollno);

	}
	fclose(fp);
}
```

Exercise 9

1. What are the main reasons of file I/O.
2. Define text and binary files.
3. Explain different types of file operations with examples.
4. What is buffer? why is it required?
5. Why do we require to open a data file? How is this accomplished? [2058/c/7-a]
6. Explain different types of file opening modes.
7. Why are fgets, fputs, fgetc and fputc used?
8. What is the main difference between fscanf and scanf?
9. Illustrate the difference between text and binary files.
10. Why are fwrite and fread functions used? Explain their arguments with examples.
11. Explain example 9.10 in detail.
12. What are stdin, stdout, stdprn, stderr, stdaux ? Why are they used?
13. Write about FILE structure.
14. Describe the three basic file operations. Write the statement used to open a file for these operations. [2059/c/7-a]
15. Differentiate between file modes r+ and w+. [059/p/7-a]
16. Write a computer program to copy the contents of one text file to another. The name of both files must be input by the user. Display appropriate message if source file does not exist. [057/c/8/,058/p/8]
17. Explain different modes of file operation and write a computer program to copy the contents
18. of one text file to another. The names of both the files must be input by the user. [057/c/8/,058/p/8,(similar)]
19. Write a program to open a new file, read name, roll_no and total_score of 24 students from the user and write to the file. After writing, display the contents of the file. [060/p/8-b]
20. Write a program to open a new file. Read the name, address and telephone of ten persons from the user and write then to the file. After writing display the content of the file. [059 Chaitra/7-b]
21. Write a program to open a new file, read name, roll number, and total score of 50 students from the user and write to the file. After writing, display the content of the file. [062/p/p/15]
22. Write a program, taking care of all the possible error conditions that may occur, to open a new file, read name, roll number, address and date of birth of students until the user says "no". After reading the data, write it to the file then display the content of the file. [2063/b/12,2063/p/11]

a macro with a arguments is known as a macro call which is analogous to function call. Example 10.2.a illustrates the concept of argumented macro substitution and 10.2.b shows the same thing using function.

Example 10.2.a: Write a program to find the area of a triangle using macro definition.

```
#include<stdio.h>
#define AREA(r) (3.14 * r * r)
void main(void)
{       float radius;
        float area;
        printf("Enter radious:");
        scanf("%f",&radius);
        area=AREA(radius);
        printf("Area=%f",area);
}
```

Output
Enter radious:12.5
Area=490.625000

Example 10.2.b: Function version of example 10.2.a.

```
#include<stdio.h>
float AREA(float);
void main(void)
{       float radius;
        float area;
        printf("Enter radious:");
        scanf("%f",&radius);
        area=AREA(radius);
        printf("Area=%f",area);
}
float AREA(float rad)
{
        return 3.14*rad*rad;
}
```

Output
Enter radious:12.5
Area=490.625000

In example 10.2.a, wherever the preprocessor finds the phrase AREA(radius) it expands it into the statement (3.14*r*r). r in the macro template AREA(r) is the argument that matches the r in the macro expansion (3.14*r*r). The statement AREA(radius) in the program causes the variable radius to be substituted for r. The statement AREA(radius) is equivalent to: (3.14* radius*radius). After this substitution, this source code is passed to the compiler to compile. The same thing can be done using function(see chapter 6) which is shown in example 10.2.b. Then a big question arises here : What is the difference in functions and macros. Which is shown in table 10.1.

Table 10.1 Differences between functions and macros

Functions	Macros
In function call, the control is passed to a functions along with certain arguments. The required calculation is done in the called functions and the final valued is returned to the calling function.	In macros call, the preprocessor replaces the macro template with its macro expansion.
Suppose, if we call a function 50 times in a program, control is passed to the function and return back but the size of program remains same. Passing control to and returning back form the called function is a time consuming process so the program execution become slower.	Suppose, if we use a macro 50 times in a program, macro extension must be done at 50 places, which will increases the size of the program.
If macros are very large and very often used then the macros can be replaced with functions.	If functions are very short and very often used then the macros can be replaced with functions.

Some library functions have macro versions. For example, **we have studied getchar as a function but it is a macro version of function getche.** We can easily identify functions and macros using the help of Turbo C or Turbo C++ compiler. For this, go to **Help→ Index→getchar** and we can see that ***getchar is macro that gets a character from stdin.*** **But in case of Help→ Index→getche**, we can see that ***gets a character from console and echoes to the screen.*** Similarly, we can find other functions having macros versions.

Some other useful examples of argumented macro definition are:

```
#define MAX_VALUE(x,y) (((x)>(y))?(x):(y))
#define STRCMP(str1,str2) (strcmp((str1), (str2))>0)
```

10.2.3 Nested macro substitution

We have already known about nesting of control statements. Similarly, one macro can be used in definition of another macro. Which is shown in example 10.3.

Example 10.3: Illustration of nested macro.

```
#include<stdio.h>
#define N 5
#define LOOP for(i=0; i<N; i++)
void main(void)
{
        int i, arr[N], sum=0;
        float average;
        LOOP
        {
                scanf("%d", &arr[i]);
                sum=sum+arr[i];
        }
        average=sum/(float)N;
        printf("Sum=%f",average);
}
```

In this example, macro N is used inside the macro LOOP.
Some other examples of nested macro definitions are:

Chapter 10
Features of C preprocessor

10.1 Introduction

We have briefly discussed about preprocessor in section 3.11. The programmers use the preprocessor tools to make his program easy to read, easy to modify, portable and more efficient. The different stages of a C program development from hand written to executable program are shown in figure 10.1. See the position of preprocessor with its input and output. It operates under the control of what is known as preprocessor command lines or directives. The preprocessor is a program that processes the source code before it passes through the compiler. The code is examined by the preprocessor for any preprocessor directives before passing it to the compiler. If there are any, appropriate actions are taken and then the source program is handed over to the compiler. Preprocessor directives begin with a # (hash) symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of the program before **main** function. The categories of preprocessor directives in the following sections are:

- Macro substitution directives
- File inclusion directives

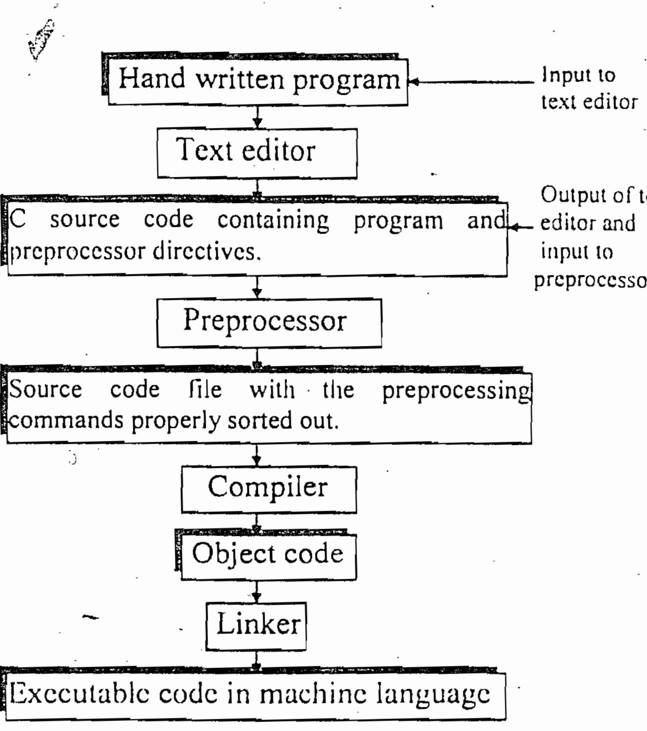


Fig. 10.1 Steps of a C program development

10.2 Macro substitution directives

It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. This process performs the task under the direction of #define statement. They take the general form as follows:
#define identifier string

Macro substation directives are of three types:

- Simple macro substitution
- Argumented macro substitution
- Nested macro substitution

10.2.1 Simple macro substitution

Simple string replacement is used to define constants. Example 10.1 illustrates the simple macro substitution.

Example 10.1: Illustration of simple macro substitution (expansion).

```
#include<stdio.h>
#define n 5
void main()
{
        int i;
        int arr[n],sum=0;
        float average;
        for(i=0;i<n;i++)
        {
                scanf("%d",&arr[i]);
                sum=sum+arr[i];
        }
        average=sum/(float)n;
        printf("sum=%f",average);
}
```

Some other useful examples of simple macro definitions are:

```
#define PI 3.1415926
#define COLLEGE "KEC"
#define g_SQUARE (9.8*9.8)
#define SUM (50+30)
#define NOT_EQUAL !=
#define LOOP for( i=0; i<30; i++)
```

10.2. 2 Argumented macro substitution

The Preprocessor permits us to define macros in more complex and useful forms. The general form is as follows:
#define identifier(a1,a2,a3.......an) string
Where a1,a2,.......an are formal macro arguments which are analogous to formal arguments in a function definition. There is no space between identifier and opening parenthesis. String behaves like a template. Subsequent occurrence of

```
#define SQUARE(side) ((side)*(side))
#define CUBE (side)    (SQUARE(side)*(side))
```

10.2.4 Undefining a Macro

When we want to restrict the definition of a particular part of the program, the macro can be undefined using the statement **#undef identifier.** For example, macro N in example 10.3 can be undefined as follows:
#undef N

10. 3 File inclusion directives

This directive causes one file to be included in another. The file contains functions and macro definitions. We have been using this preprocessor command from third chapter. The general form of the preprocessor command for file inclusion looks like:
#include "filename"
Where filename is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of the filename in to the program. The file can be included in two ways:
Using double quotation mark e.g., **#include "myfile.c"**
Using angle bracket e.g., **#include<myfile.c>**
In the first way, the command looks for the **myfile.c** in the current directory and then in the standard directory. But in the second case, the include command would look for the file **myfile.c** only in the standard directories.

Why file inclusion is required?

- It is a good programming practice to keep different sections of a large program separately. So, large programs can be divided into several different files. Each file contains a set of related functions. These files are required to include at the beginning of main program file.
- Generally we need some functions or macro definitions almost in all the programs. These commonly needed programs and macro definition can be stored in a file. Which can be included in every program that we write.

Nesting of included file is allowed i.e., an included file can include another file. But a file can not include itself. It is common for the files, which are to be included to have a **.h** extension. This extension stands for header file because it contains statements which when included go to the head of the program.

Exercise 10

1. Explain the steps of a C program development with a diagram. Highlight the position and role of preprocessor with its input and output.
2. What are different types of macro substitution directives? Explain with suitable examples.
3. Differentiate between macros and functions. Write their advantages and disadvantages. [058/p/2-a]
4. Discuss about file inclusion directives.
5. There are some other preprocessor directives. Search and discuss them with examples.

Bibliography

1. Peter Norton, *Introduction to Computers*, Tata Mc Graw Hill, New Delhi, 2002
2. Alexis Leon and Mathews Leon, *Introduction to Computers*, LeonTECHWorld, India, Chennai, 1999.
3. Deitel and Deitel, *C how to program*, Pearson Education, New Delhi, 2001
4. Balagurusamy, E, *Programming in ANSI C*, Tata Mc Graw Hill, New Delhi,1992
5. Balagurusamy, E, *Object-Oriented programming with C++*, Tata Mc Graw-Hill, new Delhi, 2001
6. Byron Gotterfried, *Programming with C, Tata Mc Graw-Hill*, New Delhi, 2000
7. Yashavanta Kanetkar, *Let Us C*, BPB Publication, New-Delhi, 2004
8. K R Venugopal and Sudeep R Prasad, *Programming with C*, Tata Mc Graw-Hill, New Delhi, 2004
9. Balagurusamy, E, *Numerical Methods*, Tata Mc Graw-Hill, New Delhi, 2000
10. Al Kelly and Ira Pohl, *A Book on C*, Addison-wesley, California, 2000
11. V. Rajaraman, *Computer programming in C*, Printice-Hall of India, New Delhi, 2000
12. Satyendra Sha, *C programming, OM publication*, Kathmandu, 2005.

Reference links

1. http://www.cprogramming.com
2. http://www.imada.sdu.dk
3. http://en.wikipedia.org/wiki/C_programming_language
4. http://www.cs.cf.ac.uk/Dave/C/
5. http://www.howstuffworks.com/c.htm
6. http://www.le.ac.uk/cc/tutorials/c
7. http://www.programmersheaven.com/zone3/index.htm
8. http://www.hermetic.ch/cfunlib.htm
9. http://cplus.about.com
10. http://www.oreilly.com/pub/topic/cprog
11. http://www.amazon.com/C-Programming-Language-2nd/dp
12. http://einstein.drexel.edu/courses/CompPhys/General/C_basics/c_tutorial.html
13. http://gd.tuwien.ac.at/languages/c/programming-bbrown/default.htm
14. http://www.apl.jhu.edu/~paulmac/intro-c++.html
15. http://www.delorie.com/djgpp
16. http://www.coronadoenterprises.com/tutorials/c/index.html
17. http://www.deitel.com/C