# Chapter - 1

# INTRODUCTION TO COMPUTER SYSTEMS AND HARDWARE

## 1. (1). Introduction of Computer :

**Computer :** It is an electronic device which processes given data to derive the required and useful information. During the processing the computer has to perform various functions like

(i) Accepting Instructions & data from the user.

(ii) Performing various arithmetic and Logical operations as per Instructions given.

(iii) Presenting the Information or Output to the user.

### Characteristics of a Computer

The Characteristics which make computer indispensable are

**1) Speed :-** The computer is able to process the date and give the output in fractions of seconds such that required information is given to the user on time enabling the user to take right decisions on right time. A powerful computer is capable of executing about 3 million calculations per second.

**2) Accuracy :-** Inspite of its high speed of processing, the computers accuracy is consistently high enough which avoids any errors. If it all there are errors, they are due to errors in instructions given by the programmer.

**3) Reliable :-** The output generated by the computer is very reliable, but it is reliable only when the data, which is passing as input to the computer and the program, which gives instructions are correct and reliable.

**4) Storage Capacity :-** The computer has a provision to store large volumes of data in the small storage devices, which have capacity to store huge amounts of data and help the retrieval of data an easy task.

**5) Versatile :-** The computer perform three basic operations

1) It is capable to access and accept information through various input-output devices from the user.

2) It performs basic Arithmetic and Logic operations on data as desired.

3) It is capable to generate the desired output in the desired form.

**6) Automation :-** Once the instructions fed into computer it works automatically without any human intervention until the completion of execution of program until meets logical instructions to terminate the job.

**7) Diligent :-** The computer performance is consistent even to all extent of more than 10 million calculations, it does each and every calculation with same speed and accuracy.

**Limitation of Computer :**

1) Computer does not work on itself, it requires set of instructions to be provided, else computer (Hardware) is waste.

2) Computer are not intelligent, they have to be instructed about each and every step which they have to perform

3) Computers cannot take decisions on its own, one has to program the computer to take an action if some conditional prevail.

4) Computers, unlike humans cannot learn by experience.

**1. (2). The Computer Generations :** The development of computers has followed different steps in the terminology used and these steps of technological differences are called as GENERATIONS in computer terminology. There are totally five generations of computers till today.

**1. First Generation Computers (1950's) :**

These computer which used vaccum tubes (valves) as major electronic component the advantage of vaccum tubes technology is that it made the advent of Electronic digital computer vaccum tubes were only electronic devices available during those days which made computing possible.

**2. Second Generation Computers (1960's) :**

With the development of Transistors and there use in circuits, magnetic core for memory storage, the vaccum tubes are replaced by transistors to arrive at 2nd generation of computers. The size of transistors is much smaller when compared to vaccum tubes, they consume less power, generated less heat and faster and reliable.

**Advantages :** (1) Size of Computer has come down as well as power consumption.

(2) The Cost of Computer reduced.

**Third Generation Computers (1970's) :**

The development of silicon chips, the third generation of computers came into existence. These computers are used Integrated Circuits (IC's) of silicon chips, in the place of transistors. Each of these IC's consist of large number of chips in very small packages.

**Advantage :** The size of computers, cost, heat generation and power consumption decreased to great extent, speed and reliability increased as

compared to previous generation. These machines used IC's with large scale Integration (LSI).

**Fourth Generation (1980's) :**

The Computers belonging to these generations used Integrated circuits with Very Large Scale Integration (VLSI).

**Advantages :** 1) These computers have high processing powers, low maintenance, high reliability and very low power consumption.

2) The size & cost of computers come down drastically

**Fifth Generation (Late 1990's) :-**

These computers use optic fibre technology to handle Artificial Intelligence, expert systems, Robotics etc., these computers have very processing speeds and are more reliable.

**1. (3). Classification of Computers :** Computers in general can be classified into major categories based on.

(a) According to the purpose of the computer.

(b) According to the operation size of computer.

**(a) Classification as per purpose of the computer :**

Classification of digital computer as per the purpose of their use

1. General purpose digital computers.

2. Special purpose digital computer.

**1. General Purpose digital computers :** These computers are theoretically used for any type of applications. These computers can be used in solving a business Problem and also used to solve mathematical equation with same accuracy and consistency. Most of the computer now are general purpose digital computers. All the P.C's, which have become household affair.

**2. Special Purpose Computers :** These digital computer are designed, made and used for any specific job. These are usually used for those purposes which are critical and need great accuracy and response like Satellite launching weather forecasting etc.

**(b) According to size and capabilities**

1. Super Computers

2. Main frame Computer

3. Medium scale Computer

4. Mini Computers

5. Micro Computers.

**(1) Super Computers :** These computers are characterised as being the fastest, with very high processing speed, very large size, most powerful. There are widely used in complex scientific applications like processing geological data, weather data, genetic engineering etc.

These computers with 16 microprocessors, will recognize words upto length 64 bits and more. The speed of calculation is upto 1.2 billion instructions/second, and they can take input from more than 1000 individual work station.

Ex. PARAM developed in India.

**(2) Mainframe Computers :** These Computer will has capability to support many peripheral devices and terminals, which can process several Million Instructions / second (MIPS), as well which support 1000 remote systems, these computers are mostly used for Railway reservation etc.

**(3) Medium Scale (size) Computers :** These computers are mini versions of mainframe computers, they are relatively smaller than mainframes and have less processing power than Mainframes their processing speed support upto 200 remote systems.

**(4) Mini Computers :** These computer are smaller and less experience than Mainframe and medium sized computers. They are relatively faster it can support about 10-20 user terminals. These computers are generally easy to use. They can handle database, statistical problems Accounting etc.

**(5) Micro Computer :** These are the mostly used category of computes called as personal computers (PC's). The word Micro suggests only the size, but not the capacity. They are capable to do all input - output operations. They can also be connected to peripheral devices.

## 1.4 Anatomy of Computers

Major Components of Computer:-

Computer is basically composed of essentially the following:

1. Hardware
2. Software

1. **Hardware:** Computer hardware is the collection of various physical components of the computer, like the computer itself, the input-output

devices. This computer Hardware performs some operations like addition, Subtraction, data transfer, control transfers and simple tests.

2. **Software:** Software is set of instructions usually termed as programs which are required for processing activities of the computer. These set of programs can be used to solve real world problems. The capability of computer depends upon the software components.

1.5 **Block Diagram of Computer**(CPU):-

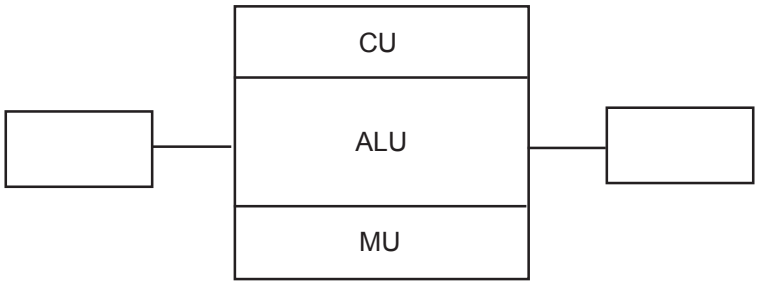



Figure 1

**Input Unit:** The process of sending the data and Instructions for the processing through some suitable devices such as Keyboard, Mouse etc. is called Input. The devices translate the data from human understandable form into electronic impulses which are understood by the computer.

**Central Processing Unit (CPU):-** Once the data accepted it fed in to Central Processing Unit before the output is generated as data has to be processed, which is done by CPU. This unit of the computer is the brain of computer system, which does all the processing, calculations, problem solving and controls all other functions of all other elements of the computer. The CPU consists of the following three distinct units namely.

1. Memory Unit
2. Control Unit
3. Arithmetic and Logic Unit

1. **Memory Unit**: Which holds the data in in terms of Program and files. The data stored can be accessed and used whenever required by the CPU for necessary processing. This memory unit is usually referred as primary storage section. The units in which memory unit is measured are known as BYTES.

BYTE is the space required to store 8 characters or alphabet or digits

to any other special character.

| | | |
|---|---|---|
| 1 Byte | = | 8 Bits. |
| 1 Kilobyte | = | 1024 Bytes |
| 1 Megabyte | = | 1024 Kilobytes |
| 1 Gigabyte | = | 1024 Megabytes |
| 1 Terabyte | = | 1024 Giga bytes |

Where Bits are spaces required to store one Binary digits i.e. either 0 or 1.

2. **Control Unit :** This unit which coordinates all the activities of each and every element of computer. It decodes the instructions given by various users and it sends commands and signals that determine the sequence of various instructions. Through this unit does not process data but it acts as the central system for data manipulation, as it controls the flow of data to and from the main storage.
3. **Arithmetic and Logic Units:-** This unit performs arithmetic operations such as addition,substraction,multiplication and division. It also does Logical Operations such as comparison of numbers etc. Thus this unit helps by processing data and taking logical decisions.

**Output: -** The processing of extracting the data from CPU through some suitable devices is called Output. The common used output devices are VDU, Printers, Plotter, magnetic media like floppy, hard disks etc.

**Types of Memories:-** The memory used by computers can be classified into various types. Some of the commonly available memory types are

1. **Magnetic Core Memory:** This is the memory which uses small magnetic cores with wires running through then and electric current which flows through these wires, which generates magnetic field. Depending on direction of current and magnetic field, data is represented. The semiconductor based memories are RAM,ROM.
2. **Random Access Memory(RAM):-** While a program being is executed it requires data, such data is stored in RAM,. The data stored in RAM is volatile i.e. when there is a power failure the data in the RAM is lost. Whenever new data is stored in RAM the previous data is erased.
3. Read on Memory (ROM): Which contain the Bootstrap Loader Programs, which loads the Operating System(O.S) into memory. As this program is stored in ROM, it starts as soon as the computer is switched on and makes the computer ready to load OS program in the memory. The contents of ROM cannot changed but for

specialized use contents of ROM can be re-programmed using special circuits.

4. **Cache Memory:** This is very high speed memory which is used to store portion of a program from main memory temporarily. The Cache memory very expensive, but it is very fast,. The data is transferred automatically between the buffer and primary storage, which lies between main memory and Microprocessor.

**Communication Paths:**

The CPU consists of Memory Unit, Control Unit, and Arithmetic and Logic Unit (ALU). The Communication links between these three units are called buses. The buses are the electrical paths for data to flow from point to point in a circuit.

The Buses for Communication in CPU are 1. Control Bus 2.Address Bus 3. Data Bus

1. **Control Bus: -** The Path for all controlling and timing functions sent by the control unit to other units of CPU.
2. **Address Bus:** - This Path is used for locating the address of the memory location where the next instruction to be executed or the next piece of data is found. This is used for data retrieval storage and manipulations where address of storage locations are required.
3. **Data Bus:-** This is the path on which actual data transfer takes place.

**SOFTWARE:** - Collection of Programs are Software. The Software is those components which make the task of a user complete, as it is that component which helps a user to give set of instructions. The Software is categorized on the basis of functions they are

1. System Software 2 .Application Software

1. **System Software:-** Which controls all processing activities and makes sure that the resources and the power of the computer are used in most efficient manner . The System software can categorized in to

    1. **Operating System**
    2. **Language Processors ( Assemblers, Compilers, Interpreters)**
    3. **Device Drivers**

### 4. Utility Programs etc.

The System software as a package which consists of various programs required for the functioning of the computer itself. It is also needed for translation, loading, controlling and running of the program.

The system software basically serves TWO major purposes.

a. It controls the execution of programs on the computer.
b. It helps the development of software.

1. Operating System :- Which is set of programs that controls and support hardware and provide various services which are used for better performance of computer . The major functions of O.S are
    1. It assigns processors to tasks
    2. It manages memory and other storage areas
    3. It acts as a command interpreter
    4. File management
    5. Input-Output Management
    6. Establishing data security & Integrity
    7. Maintains account of processor time for billing purposes.
    8. Provides data and time services etc.

   The Operating systems can be classified as Single user and Multi-user (number of users working on it at a given point of time) and Multitasking operating systems,

| | |
|---|---|
| Single user Operating systems | - MSDOS |
| Multi-user Operating System | - UNIX, Linux etc |
| Multitasking Operating System | - Windows |

### 2. Language Processors:-

**Translators:** As the computer accepts digits and characters as input, such input to be conversed. Into machine language. The Software which makes conversion possible and increases the productivity of the programmer are called Translators.

There are three basic types of translators they are

**i. Compilers** **ii. Interpreters** **iii. Assembler**

i. **Compilers: -** A compiler is a software that will convert the high level instructions into equivalent Machine level

instructions at a time.

ii. **Interpreters:-** A Interpreter is a software that will convert the high level instructions into equivalent Machine level instructions step by step.

iii. **Assembler:-** Assembler is a software that will convert the high level instructions into equivalent Machine level instructions but here, the input programs are in assembly language. The output is in machine language.

**2. Service Utilities:**

These utilities are those software programs which are provided by the manufacturer of various hard wares to perform specific tasks which are common to all data processing installations.

The commonly available service utilities are:
a) Device Drivers
b) Directory and File management Utilities.

a) **Device Drivers:** These are special programs which are used to enhance the capability of operating systems, so that it can support many input/output devices like mouse, joysticks, printers etc.
b) **Directory and File management Utilities:** These are those software programs which are used to manage data.

**II. APPLICATION SOFTWARE:**

This is that software which serves specific purposes and allows the user to create applications which are for a given purpose like financial accounting, payroll examinations, human resources management etc.These software can be further classified, depending upon the source of development as well the users:

**i) Pre written application software.**
ii) **User written application software.**

**i) Pre written application software.**

These are those software packages which are developed by group of people or an individual to be used by others.
The most commonly available prewritten application software are:

i) Word processing software
ii) Electronic spreadsheets
iii) Database management software
iv) Graph generator

v) Report generator
vi) Communication software

i) **Word processing Software :** These are those software, which usually automate the day today documentation work of an organization.

ii) **Electronic spreadsheets :** Electronic spreadsheets are like sheets are of paper with rows and columns. Electronic spreadsheets allow numbers, characters, formulas and all other types of data which has to be entered in a tabular form into rows or columns. Some of the significant advantages are:-

a) It provides flexibility and it is easy to incorporate changes.
b) Automation of various calculations.
c) Flexibility to size and resize columns.

Some of the most commonly available and used spreadsheets are Lotus-1, 2, 3, MS-Excel, Super Calc etc.

**Database management software:** Database is an organized collection of data, which is logically related.

**Graph generator:** As graphical representation of data is much more easy to understand as well is appearing, when the data presented in graphical from like Bar graphs, Pie charts, Line graphs etc. is called Graph generator. Some of the graphics generators are Lotus 123, Oracle graphics, MS Excel etc.

iii) **Report generator:** There is always a need in day to day operations that timely reports of various activities have to be used, so as to depict, a situation or helping decision making.

iv) **Communication Software:** Communication is the essence of all operations. Effective commu8nication i.e., sending and receiving data plays a key role in the smooth running of any organization.

## 1.6 Number System

Data is usually combination of Numbers, Characters and special characters. The data or Information should be in the form machine readable and understandable for his reason the data has to be represented in the form of electronic pulses.

The data has to be converted into electronic pulses and each pulse should be identified with a code. For these reasons, the data is converted in to numeric format first, by using ASCII, where each every character,

special character and keystrokes have numerical equivalent. Thus using this equivalent, the data can be interchanged into numeric format. For this numeric conversions we use number systems, each number system has radix or Base number , Which indicates the number of digit in that number system.

Basically Number system is divided in to

1. Binary Number System
2. Octal Number system
3. Decimal Number system
4. Hexa- Decimal Number System

1. **1. Binary Number system :-** This system which represents only two types of digits i.e 0's and 1's, so the radix number is this system is 2. The advantage number system is that, as there are only two distinct digits one can use only two electronic pulses to represent data. The two pulses that are used to represented the digits of this number system i.e. 0 and 1 are

- Absence of pulses(current) for 0
- Presence of pulse(current) for 1

The value of the numbers is represented as power of 2 i.e. the radix of the system. These power increases with the position of the digit

| Position | $6^{th}$ | $5^{th}$ | $4^{th}$ | $3^{rd}$ | $2^{nd}$ | $1^{st}$ |
|---|---|---|---|---|---|---|
| Weight | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

Ex: Convert (10101)2 in to Decimal equivalent

$$10101 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 0 + 4 + 0 + 1 = (21)_{10}$$

**2. Octal Number System :-** This number system has a radix of 8 and the digit of this number system are 0 - 7. All the numbers are represented as the combinations of 0 - 7.The position of the number determines its value in this number system, which is power of 8 i.e.

| Position | $6^{th}$ | $5^{th}$ | $4^{th}$ | $3^{rd}$ | $2^{nd}$ | $1^{st}$ |
|---|---|---|---|---|---|---|
| Weight | $8^5$ | $8^4$ | $8^3$ | $8^2$ | $8^1$ | $8^0$ |

This number system has a concept of representing three binary digits as

one octal numbers, thereby reducing the number of digits of binary number still maintaining the concept of binary system.

**3.Decimal Number system:-** The digits of this number system 0 .. 9., the radix of this number system is 10. In this system the value of a digit depends upon position they occupy. These positions are in the multiplies of 10 i.e the radix of the system.

| Position | $6^{th}$ | $5^{th}$ | $4^{th}$ | $3^{rd}$ | $2^{nd}$ | $1^{st}$ |
|---|---|---|---|---|---|---|
| Weight | $10^5$ | $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ |

**4. Hexa Decimal Number System:-** The digits of this number system is 0 -15 (i.e the digit represented as A–10 is represented as B–11, C–13, D–14, E–15, F–15) . The radix of this system is 16. In this system we group four binary digits to represent hexadecimal numbers.

Ex : 7B72, 5C88, 4A51 etc.

**Conversion of one number system to another number system:-**

1. Binary **number system to Octal number system and voiceovers**
2. **Binary number system to Decimal number system and voiceovers**
3. **Binary number system to Hexa Decimal number system and voiceovers**
4. **Octal number system to Decimal number system and voiceovers**
5. **Octal number system to Hexa number system and voiceovers.**
6. **Hexa Decimal number system to Decimal number system and voiceovers**

**Arithmetic Operations using Binary system:-** The basic arithmetic operations which can be performed rising binary number system are:-

**a.** Addition b. Subtraction c.Multiplication d. Division

a. **Binary Addition:-** Binary addition is similar to decimal numbers. The rules involved in carrying out the binary addition are as follows

**0 + 0 = 0**
**1 + 1 = 1**
**1 + 0 = 1**
**1 + 1 = 0** with a carry over of 1.

Ex:- Add $(10011110)_2$ and $(11101101)_2$

| Sol:- | Binary | Decimal |
|---|---|---|
| | 1 0 0 1 1 1 1 0 | 1 5 8 |
| | 1 1 1 0 1 1 0 1 | 2 3 7 |
| | **1 1 0 0 1 0 1 1** | **3 9 5** |

a. **Binary Subtraction:-** Binary subtraction is also very similar to decimal susbstraction. The rules binary subtraction are

**0 + 0 = 0**
**1 + 0 = 1**
**1 + 1 = 0**
**0 + 1 = 1** with one barrow

**Ex: -** Substract $(1110110)_2$ from $(1111101)_2$

| Sol:- | Binary | Decimal |
|---|---|---|
| | 1 1 1 1 1 0 1 | 125 |
| | - 1 1 1 0 1 1 0 | - |
| 118 | | |
| | 0 0 0 0 1 1 1 | 07 |

b. Binary Multiplication:- To multiply binary numbers, the rules which have to be followed are as follows:

0 x 0 = 0

1 x 0 = 0

0 x 1 = 0

1 x 1 = 1

Ex:- Multiply $(11110)_2$ by $(100)_2$

| Sol: | Binary | Decimal |
|---|---|---|
| | 1 1 1 1 0 | 30 |
| | 1 0 0 | x 4 |
| | 0 0 0 0 0 | 120 |
| | 0 0 0 0 0 x | |
| | 1 1 1 1 0 x x | |
| | 1 1 1 1 0 0 0 | |

**d. Binary division:-** For binary division, the rules which are applicable to the decimal system has to be followed and the rules of binary addition, subtraction and multiplication have to be adhered to

Ex:- Divide $(100011)_2$ by $(111)_2$

Sol:-

```
          111 )  100011  ( 101
                 111
          ------------------
                 00111
                   111
          ------------------
                   00
```

**Binary Complements:-**
**Def:-** Replacing 0's with and 1's with 0 is called complement of given number.

Ex:- Find the complement of $(11111001)_2$

Given Number 1 1 1 1 1 0 0 1
1's Complement 0 0 0 0 0 1 1 0

**Complementary Subtraction**:- This is for subtracting binary number from given binary number One's complement of a number is arrived at by susbstracting each binary digit from 1

Ex:- Find out one's complement of $(1101001101)_2$

Sol:- To arrive at one's complement substract each digit from 1.

```
1  1  1  1  1 1 1 1 1
1  0  1  0  0 1 1 0 1
1        0  1  0  1  0 0 1 0
```

Substraction using complements:-
Substraction using 1's complement:-

Steps:- 1. Find one's complement of the number to be subtracted.
2. Add the complement to the number from which it has to be subtracted.
3. If there is carry of 1, at the end add that back to the resultant, to get subtraction's result.
4. If there is no carry i.e. o, then add o as no carry, then recomplement

it and attach a negative sign to get subtraction's result.
Ex:- Subtract 1011 from 110111 by using 1's complement.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 1 | 0 | 1 |
| + | | 1 | 1 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 0 | 0 | 0 | 1 | (add carry of 1) |
| | 1 | 0 | 0 | 0 | 1 | 0 | |

**i. Subtraction using 2's complement:-**

**2's complement = (1's complement) + 1**

Steps:- 1. Find the 2's complement of the number to be subtracted.

3. The complement which is arrived at 1st step is added to the number from which it has to be subtracted.
4. If it results in a carry of 1, ignore that to obtain the result
5. If it results in no carry , add 0. Find the 2's complement of the result and attach a negative sign to obtain the required result.

Ex:- 10100 - 10110 using 2's complement.

| | | | | | | |
|---|---|---|---|---|---|---|
| Sol:- | Given number | 0 | 1 | 0 | 0 | 1 |
| | + | | | | | 1 |
| 2"s | Complement | 0 | 1 | 0 | 1 | 0 |

Add 2's complement to 10100

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 0 | 1 | 0 | 0 |
| + | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | |
| | | | | 0 | |
| 1 | 1 | 1 | 1 | 0 | |

## 1.7 Basic Input/Output Devices

These are those devices which facilitate a user to gives input, i.e. date to the CPUto work on.

**i) Punched Cards :** These input units were used by Charles Babbage, the "Father of Computers" These punched cards are rigid pieces of paper which contain rows and columns of Numbers. These cards are kept ion a punch machine, which makes holes in the cards to represent the data.

Punched card has the following advantages:

ii. It is relatively cheaper.
iii. Both user as well as machine can read data.

ii)Punched Paper Tape: This is a continuous strip of paper on which the characters are represented by using a code which is made of combination of circular holes punched across the width of the paper.

Advantages:

i. As the paper tape is cheaper than punched cards
ii. The speed of paper tape reader being about 1100-1000 characters per second.

v) Magnetic Tape: Magnetic tapes are made of thin plastic coated with magnetic materials like iron oxide. It can be magnet5ized in either of two direction. Magnetic tape drive used to read and also to write data on the magnetic tape. The speed of the magnetic tape drive is about 5000to 2,40,000 characters per second. Advantage of this tape is it can be reused.

vi) Video Display Unit (VDU):- These are also used as input devices for computers. Are very similar to a picture tube of television. These are also know as Cathode Ray Tube.

vii) Key Boards:

This is the most common and popular input device. It is similar to the ordinary typewriter.

Key boards usually contain the following keys:-

i) Numeric Keys
ii) Character keys
iii) Function keys
iv) Special keys

vi) Mouse: This input device is used specially for Graphic User Interfaces(GUI). It contains one to three buttons.

This is used as a pointing device.

vii) Joy Stick: This is one of the popular input device, which is used for playing video games.

viii) Touch Screen: Touch screen takes input as when a finger or object cones in contact with the screen, the light beam is broken, and the location of breaking of light beam is recorded, which acts s inputs.

ix) Magnetic Ink Character Recognition: This MICR reader reads the characters which are written using special magnetic ink. These are also human readable. Advantages like: Inspite of rough handling, one can read the information with a very high degree of accuracy.

X) Magnetic Strips: These are thin bands of magnetically encoded data. These are usually sued on credit cards. The data cannot be seen and read by mere looking at the card, there here it is usually secret.

XI) Light Pen: Light is a type of pointing device that is used to choose a displayed menu option on screen for a program. This pen contains a photocell placed in a small tube.

XII) Voice Input Devices: These devices are in stage of experimenting and development, where various problems like speech should be independent of persons.

XIII) Optical Mark Reader: These devices use any of two methods, one method relies on conductivity of graphite which determines the presence of pencil mark, for this only soft pencil should be used.

**Output Devices:** Output devices can adopt different shapes and styles, which are used to give the output of the processing of data. The output can be in any form i.e. typed, printed, sketched, visible, nonviable, audio, video etc. Output devices are those peripheral devices that allow a user to retrieve information from a computer. Any Output devices can be devided in to two basic categories

1. Hard Copy 2. Soft Copy

1. **Hard Copy** : It is that form of output which can be read directly and immediately, stored to be read later. This output is on paper and which is permanent.
2. **Soft Copy:** It is magnetic or audio form of output which is usually unreadable directly and which is not long-lasting.

**Hard Copy Devices:-** These are those devices, Which produce permanent record of data on media like paper. These are relatively slow in operation, when compared to soft copy devices. Some of the popular devices are

a. Printers b..Plotters c.Photographic output devices d.Punched cards.

a. Printers : Printers can be categorized in to

1. Impact Printers
2. Non-impact Printers

a. **Impact Printers:** These are those type of printers which have direct mechanical contact between the head of the printers and paper. The impact printers are

**1.Dot Matrix Printers: -** These Printers print characters as combination of dots arranged in a 5 x 7 matrix. The speed of these printers vary from about 30 to 600 characters per second. The printing head of these printers contain vertical array of pins, which fires the selected pins against carbon ribbon or linked surface, while the head moves across the paper to form a pattern of dots representing characters. These printers can print characters in draft quality, standard quality as well near letter quality.

b. **Daisy Wheel Printers**.:- This printer is a solid font type character printer. The printing arms of the printer head look like petals of the flower. This printer is very slow,it prints about 60-90 characters per second. The hub rotates continuously, rotating at high speed and hammer strikes the appropriate character in its position. This printer is also act as bi-directional printer .
c. **Line Printers:**- These printers which print total line at a time, giving the name line printers. They can be print 1000-5000 lines per minute.

Non -Impact Printers: - These printers print without any direct mechanical contact between the head of the printer and paper. These printers can categorized in to

a. Thermal b. Inkjet c. Laser d. Electrographic e. Electro Static

**a. Thermal Printers:-** These Printers uses heat sensitive paper in dot matrix method, on which hot needles are pressed to form characters. These printers print slow and makes little noise. The disadvantage is that it is not possible to print multiple copies simultaneously.

**b. Inkjet Printers:-** These printer uses a mechanism where it sprays the ink from tiny nozzles through an electric field that arranges charged particles ink into characters. The paper absorbs the ink and it dries instantaneously. It is capable to print about 500 cps , expensive but faster and we can also have color printing, by using ink various colors.

**c. Laser Printer:-** This type of printers uses laser beams which charge the drum negatively, to which black toner powder which is positively charged is sticked. When the paper rolls by the drum, the toner powder is transferred to the paper. These printers are very fast, quality is very good and speed about is 15,000 to 20,000 lines per minute.

**d. Electrographic Printer:-** This printer writes on special paper usinf electricity. The paper is held between two electrodes. These printers can either use wet process or dry process. The speed of these printers are fairly high and is about 20,000 cps. These printers are very reliable as there are very less moving parts.

**e. Electro Static Printer:-** This printer uses a mechanism, where static electricity is used to create impressions on special paper. The charges attract toner power to the charged spots, giving the print. This printer s are very fast and are capable for plotting graphs etc. The speed is very high i.e up to 20,000 lines per minute.

**II. SOFT COPY DEVICES:-** These devices output is usually not visible directly but cab be viewed with help of computer where one can update, modify data etc. Some of the softcopy output devices are

**a. Visual Disply Terminal :-** 1. Dumb Terminal 2. Smart Terminal 3. Intelligent Terminal

**b. Visual Display Unit (Monitor) :** The screen of monitor is generally two types.

i. Cathode Ray Tube: ii. Flat-panel Display

### 1.8 Memory Units

Basically Computer Memory is divided in to Primary Memory and Secondary (Auxiliary) Memory

**Primary Memory : -** Which is purely temporary memory (i.e the data or program lies in this memory till the program is executed or power is lost ) i.e volatile

Ex. RAM,ROM ,PROM,EPROM and EEPROM etc.

**Secondary Memory :** Which is non-volatile memory and which is external to computer. It is a secondary media used for storing large volumes of data permanently for long term.

The following are the secondary storage devices:-

1. Magnetic Tapes 2. Magnetic Disks 3. Optical Disks

**1. Magnetic Tapes :-** These are external storage devices which as used as high speed input and output devices. These devices use a tape which is about 12.5 mm to 25 mm wide and about 500meters long. This tape made up of plastic like material known as mylar. This tape is wound on a spool which can be loaded onto the tape deck. This deck is connected to the CPU permanently and the information stored into or read from ,the tape by the processor. This tape is coated with a very thin layer of iron oxide, magnetic material.

These tapes are to be accessed serially . The data is stored as binary digits i.e. one and zero are represented as presence and absence of magnetic field. i.e. spots on the tape.

Advantages of using Magnetic tapes:-

1. Large amount of data can be stored in to small lengh of magnetic tape
2. The Tape is very easy and convenient to handle.
3. The tape cost is very economical
4. It is very fast and life of the magnetic tape is very high, permitting long term storage.

**Magnetic Disks: -** Magnetic disk is usually made up of plastic like material called mylar or any metallic platter coated with ferromagnetic material. The data is stored in the same way as magnetic tapes. Here read/write heads are small electromagnets which are capable to read, write or even erase the data stored as magnetic spots.

They can have varied storage capacities. These disks can be divided in to

1. Fixed or Hard Disks 2. Floppy disks.

1. **Fixed or Hard disks:-** Fixed or Hard disks is usually made from aluminum or other materials instead of mylor or other plastic like material. . Hard disk is permanently installed in the drive . This disk is usually large(about14inches). These disks also come in various sizes like 9inch,5inch etc.
2. **Floppy disks:** This new device used for secondary storage media, this small ,flexible, faster and cheap alternative for the storage purpose. This disk is made up of flexible plastic material, coated magnetic materials like iron oxide. Data is recorded in the form of invisible magnetic spots on coating. This floppy diskette comes in three basic sizes are 8 inch,5.25 inch and 3.5 inches
3. 
4. To store data in the floppy it has to formatted i.e. the floppy surface has to be divided in to tracks and sectors.
5. **Tracks:-** These are logical concentric circles which start from the edge of the floppy disk and move inwards, towards the center of the disk. The tracks are numbered.
6. **Sectors:-** The small portions of the tracks are called sectors where the data is stored physically. The amount of data stored in a sector is depends on the floppy disk. When the data is stored in a sector, if it becomes fully, then it I stored in the nest sector. The floppies can divided according to storage capacities.

1. 48 TPI- Double Sided -Double Density (DSDD)

   ex: 5.25 inch- 360KB
2. 96 TPI- Double Sided-Double Density(DSDD) Ex:- 720 KB
3. 96 TPI- Double Sided-High Density(DSHD) Ex : - 1.2 MB
4. 135 TPI- Double Sided-High Density(DSHD) Ex:- 3.5 inch - 1.44MB

Advantages of Magnetic Disks:

1. The magnetic disks provide for easy accessibility to the stored data. The time takes for access is very less.
2. The disk has longer life then tapes.
3. We can access data both sequentially, or randomly

4. We can update several on line disk records by single input transaction

3. Optical Disks:- This Optical technology is capable storage devices which can store large amount of data . This technology involves the uses of laser beams, which are highly concentrated beams of light. Using this technology several secondary storage devices can be revolutionized by the use of the following secondary storage devices based on optical technology.

1. Optical disk
2. Optical card
3. Optical tape

**1. Optical disk:-** This is similar to the laser-optical video disks which are sold for home purpose. Optical disks use same technology for storing data onto them. This disk is made of hard metal ranging from 4.72 inches to14 inches in size. These disks were initially developed and used as compact disks for video and audio applications. To store data laser beams are used to burn the microscopic “pits” to represent binary digits i.e. 0’s and 1’s. These pit patterns represent data. These disks are read using optical disks drives that use beam of lasers to read pit patterns and convert them in to bits.

These disks usually come as read only . These disks are usually used to store programs, utilities, software, data etc. so that one can read the contents and carry on required work. These are generally termed as CD ROM’s.

Erasable Optical disks:- These are the disks which are erasable and can be reused. These disks are bit expensive. We can use CD-writer to write in to these CD’s.

II. **Optical Card:-** These are similar to credit cards in size and they have an optical laser encoded strip. These cards can store about 2 MB of data. These cards can be used to storing credit records, also can be used all cash transactions as well as cheque transactions.

III. **Optical Tape:-** This is similar to magnetic tape in looks, but it differs that the data is stored using optical technology. These tapes come in cassettes, can store 8GB. These tapes are read by using Optical tape drives, These drives are capable le of holding about 128 cassettes.

## 1.9 Operating Systems

Operating System (OS) can be defined as “set of Programs that are capable to manage resources as well as the operations of a computer”.

Operating system hides the hardware from the user

The basic functions carried by the Operating system are:

1. Assigning processors for performing tasks.
2. Allocating and managing memory, and other storage area
3. Command interpretation
4. Handling job transactions.
5. Maintains internal clock
6. Scheduling of various tasks.
7. Establishing and enforcing priorities for different jobs.
8. Co-ordinating and managing peripheral devices.
9. Input-output management
10. Establishing data security and integrity.

**Operating system concepts:-** User programs interact with operating system using set of extended instructions. These instructions are called "system calls". These system calls are used to create, delete and use various software objects that are manages by the Operating systems. The following are common in any Operating system.

1. Process
2. Files
3. System Calls
4. The shell
5. Booting Process i. Boot Strap Loader
   - i. Check Programs
   - ii. Monitor Program
   - iii. Basic input/output System(BIOS) Program
   - iv. Utility Programs
   - v. File Maintenance Programs

**Types of Operating systems:-** Operating systems are basically capable to do all functions but The way of processing or approach of the systems may vary from one Operating systems to another. The following basic types of operating systems are

1. Batch Processing Systems
2. Interactive Operating systems
3. Multi-Programming Operating Systems
4. Multi-Processing Operating Systems
5. Multitasking systems
6. Multi-user Operating systems
7. Virtual storage systems

Basing on the features of the operating systems and the interface provided they can be classified as

11. **User friendly Operating system :-** These Operating systems provide a pleasant and easy to work environment, they are usually graphical based, where the various options are represented as icons, menus etc. Windows 95, Windows 98, Windows-NT are some of the user friendly Operating systems.
12. **2. Programming friendly Operating systems:-** The programmer is anyone who has fair knowledge about programming concepts and he is expected to know the computer in a detailed manner. The programmer much concerned about various utilities, functionlities, flexibility and powerful environment. UNIX, XENIX,LINUX etc. are some of the character

based interfaces which are very powerful and programming friendly.

**Character User Interfaces( CUI):-** This user interface is character based, one gets prompts in characters and a user has to type the characters. These were the interfaces which enabled the programmers to do work easy, but they it is difficult to work in as we have to type every instruction and commands . MS-DOS,UNIX,LINUX etc. are some of the most popular character based operating systems.

**Graphical user Interface(GUI):-** This interface is graphical i.e. there are pictures, graphical objects, images, menus etc . which have specific functionality's. This interface provides very pleasant environment to work on it. Here almost every action is automated, at click of a button can get desired action done. Windows95,Windows 98, Windows-NT are some of very popular GUI based Operating systems.

**Features of Operating Systems:-** The major features of an Operating System are

I . Processor Management

II. Device Management

III. Memory Management

IV . Security Management

V. File Management

H H H

## CHAPTER - 2

# INTRODUCTION TO PROBLEM SOLVING TECHNIQUES

## 2.1 ALGORITHM

Algorithm is a method of representing the step-by-step logical procedure for solving a problem. An algorithm is a recipe for finding the right-answer to a problem or to a difficult problem by breaking down the problem into simple cases.

According to D.E. Knuth, a pioneer in the computer science discipline, an algorithm must possess the following properties

(i) Finiteness : An algorithm must terminate in a finite number of steps

(ii) Definiteness : Each step of the algorithm must be precisely and unambiguously stated

(iii) Effectiveness : Each step must be effective, in the sense that it should be primitive (easily convertable into program statement) can be performed exactly in a finite amount of time.

(iv) Generality : The algorithm must be complete in itself so that it can be used to solve all problems of a specific type for any input data.

(v) Input/Output : Each algorithm must take zero, one or more quantities as input data produce one or more output values.

An algorithm can be written in English like sentences or in any standard representation Sometimes, algorithm written in English like languages are called Pseudo Code

**Example :**

1. Suppose we want to find the average of three numbers, the algorithm is as follows

   Step 1 Read the numbers a, b, c

   Step 2 Compute the sum of a, b and c

   Step 3 Divide the sum by 3

   Step 4 Store the result in variable d

   Step 5 Print the value of d

Step 6 End of the program

2. Find the average of n inputted numbers

| | | |
|---|---|---|
| Step 1 | Read the total number of ovales,n | |
| Step 2 | Assign the value 9 to count | (count = 0) |
| Step 3 | Assign the value 0 to sum | (sum = 0) |
| Step 4 | If the value count is equal to the value n go to the step 9 | (if count = n go to step 9) |
| Step 5 | Read the value, a | |
| Step 6 | Add the value of a to sum and store into sum variable | (sum = sum +a) |
| Step 7 | Increment the value of count by 1 | (count = count +1) |
| Step 8 | Go to step 4 | |
| Step 9 | Compute the division of sum and n and store the result into avg variable | (avg = sum / n) |
| Step 10 | Print the value of avg | |
| Step 11 | End of program | |

The above algorithm terminates after some time, which explains the property of finiteness. Action of each step is precisely defined. In this example, each step requires simple operations such as reading, assigning, comparing, printing and so on. This explains the feature of definiteness and effectiveness. Total number of entries and their values will be the input for the algorithm where as the average of the entered values will be the output. Formally, an algorithm can be defined as an ordered sequence of well-defined and effective operations that, when executed, will always produce a result and eventually terminate in finite amount of time.

When a problem is to be solved or some complex task is to be accomplished, the most natural way is to break the problem (or task) into small problems, which can be solved easily. Consider a shopping example where the task is to purchase the required items. This can be achieved by doing the following tasks.

1.1 Going to the shop

1.2 Selecting and purchasing the required items

1.3 Returning back

Each of these tasks can be further divided into sub-tasks.

1.1.1 Go to the bus stop

1.1.2 Board the appropriate bus

1.1.3 Get the ticket

1.1.4 Reach the destination

The second task can be further divided into following sub-tasks.

1.2.1 Select the item and respective quantities

1.2.2 Check for the total items

1.2.3 Make payment

1.2.4 Collect the items purchased

The third task can be classified into following sub-tasks

1.3.1 Goto the bus stop, which is near to the shop

1.3.2 Board the appropriate bus

1.3.3 Get the ticket

1.3.4 Reach residence

The decomposition of one task into smaller sub tasks can be represented as...

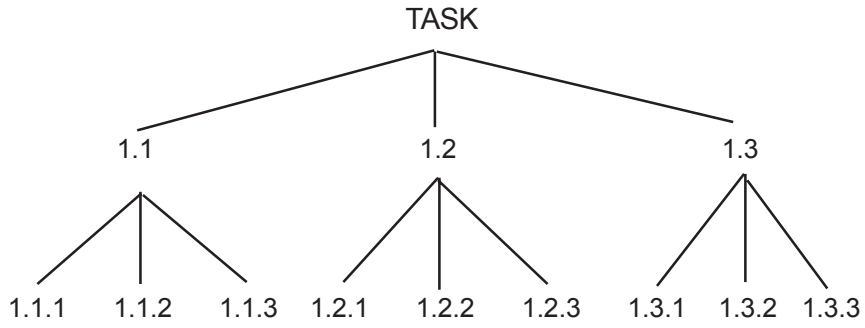



Figure 4.4 :

The process of breaking each problem or task into sub-tasks may continue until-the problems or tasks reached the lowest-levels that can be easily solved or accomplished. This is referred to as top-down design technique.

Each time the problem is divided into smaller problems, which are easier to solve than the original problem. Therefore, top down designing is also called step-wise refinement. This technique makes programs easy to write, read, modify & maintain.

Example :

To find the sum of the series $1+\frac{X}{1!}+\frac{X^2}{2!}+\frac{X^3}{3!}+.....+\frac{X^n}{n!}$

The steps for top-down design are

1. Input value of x
2. Input number of terms

2.1 get number of term

2.2 Check whether the number of terms is a positive integer or not

3. Find sum of series
   - 3.1 Find terms
     - 3.1.1 Find the factorial of specified integer
     - 3.1.2 Find power of x
     - 3.1.3 Compute division of power of x by factorial of specified integer
   - 3.2 Find sum of specified number of terms

**4. Print result**

Time and space are two important parameters forevaluating the efficiency of an algorithm i.e.,

(i) How fast does an algorithm work

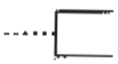
(ii) How much storage space does it require

Generally, the need to analyse efficiency arises, when alternative algorithms are compared for a given problem or when the problem is complex and/or very large.

## 2.2 FLOWCHART

Flowchart is diagrammatic representation of an algorithm. it is constructed using different types of boxes and symbols. The operation to be performed is written inside the box. All symbols are connected by arrows among themselves to indicate, the flow of information and processing.

Following are the standard symbols used in drawing flowcharts.

| | | | |
|---|---|---|---|
| Oval |  | Terminal | start/stop/begin/end symbol |
| Parallelogram |  | Input/ouput | Making data available for processing (input) or recording of the processed information (output) |
| Document |  | Printout | Show data output in the form of document |
| Rectangle |  | Process | Any processing to be performed. A process changes or moves data. An assignment operation normally represented by this symbol. |
| Diamond |  | Decision | Decision or switching type of operations that determines which of the alternative path is to be followed |
| Circle |  | Connector | Used to connect different parts of flow chart. |
| Arrow |  | Flow | Joins two symbols and also represents flow of execution |
| Bracket with broken line |  | Annotation | Descriptive comments or explanations |
| Double sided rectangle |  | Predifined process | Modules or subroutines specified elsewhere |

1. Find out the average of a n numbers

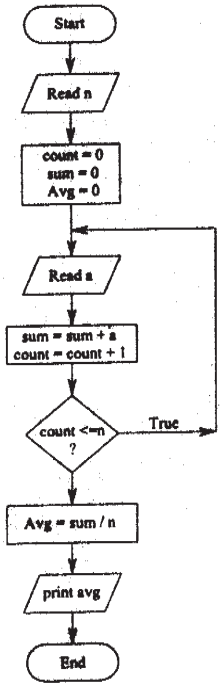



FIGURE 4.5:

2. Find out the factorial of any given nuber

NI=1*2*3*....*N-1*N if N>0
=1 if N=0

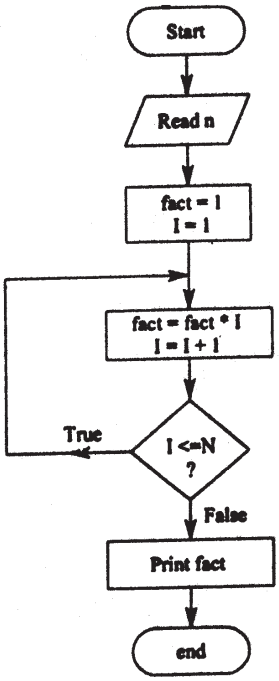



FIGURE 4.6:

3. Generate and print the firtst N terms of Fibonacci Series, where N is greater than or equal to 1.

The Fibonacci series is 0,1,2,3,5,8,13.....
The Fibonacci relations is represented as

f(n)= f(n-1)+f(n-2) where f(0)=0 and f(1)=1

Each number, after the first two, is derived from the sum of its two nearest predecessors.

*Problem understanding*

In the Fibonancci series only the first two numbers (0 and 1) are define. After that, each number is the sum of the previous two numbers. This can be illustrated as

New number = Sum of last Two numbers

1 = 0+1
2 = 1+1
3 = 1+2
5 = 2+3
8 = 3+5

Development of algorithm

Define three variables sum, A and B

Sum is used to store the next number

A is used to store the number before the last number

B is to store the last number

Sum = A+B

Here the values of A and B keep on changing

1 = 0+1
2 = 1+1
3 = 1+2
5 = 2+3
8 = 3+5
13 =5+8

It can be observed that the next value of B is the previous value of sum and the next value of A is the previous Value of B. For example, when sum is 13 then A and B are 5 and 8 respectively. This process of exchainging the value of sum, A and B and addition of A and B to get the value sum continues depending on how many numbers have to be produced.

Description of the algorithm

Step 1 Read the total number of required value in the Seris, *N*
Step 2 Initialize the value of *A* with 0 and B with 1.
Step 3 Displlay the values *A* and *B*
Step 4 Initialize the counter *1* to 2
Step 5 Compute the next value by adding *A* and *B* and store into sum
Step 6 Display the value of sum
Step 7 Transfer the value of *B* to *A*
Step 8 Transfer the value of sum to *B*
Step 9 Increment the counter *1* by 1
Step 10 If the value of *1* is less than or equal to *N,* then go to step5
Step 11 End of the program

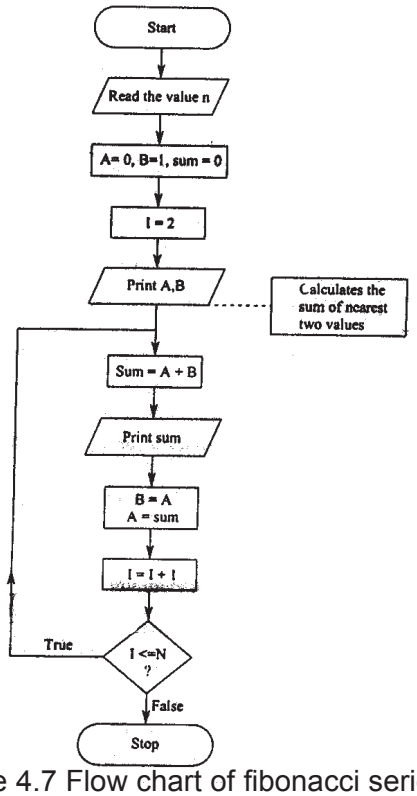



Figure 4.7 Flow chart of fibonacci series

Important points in drawing flowcharts

1. Flow chart should be clear, neat and easy to follow
2. Flow chart should be logically correct
3. Flow chart should be verified for itss validity with some test data

Limitations of flow charts

1. Flow charts are difficult to modify. Re-drawing of flowchart may be necessary
2. Translation of flowchart into computer program is always not easy.

Advantage of flowcharts

1. Logical of program is clearly represented
2. It is easy to follow logically the flow chart

## 2.3 HEURISTICS

Heuristic refers to the process of knowing by trying rather thaan by following some preestablished formula. This is in contrast to Algorithmic approach.

*Heuristic approach usually includes two usages:*

**(a) Trial-and-error Learning :** Describing an approach to learning by trying without necessarily having an organized hypothesis. Or, its is also

the way of proving that the results proved or disproved the hypothesis.

**(b) Rule-of Thumb :** Describes an approach that uses the general knowledge gained by experience.

Heuristic are used in complex large programs such as weather forecasting, chess game and stock market prediction. However, unlike algorithms, heuristic does not always provide best possible solution. But it hleps in guessing the solution fast.

## 2.4 PROGRAMMING FEATURES

Programs solve problems by manipulating information. In general the basic elements of programming are :

**1. Datatypes :** These are storage structures that use constants, variable, characters, strings, addresses etc.

**2. Operations :** used to manipulate the data such as assign one value to another, combining values (add, subtract, etc.), compare values (equal, less than, greater than etc.) etc.

**3. Input and output :** Reading values from an input device such as keyboard, disk, etc. and writing information to the output device such as screen (or monitor) disk, etc.

**4. Conditional Execution :** Executes an instruction or a set of instructions, which depends on a specified condition.

**5. Loops (iterations):** Executes an instruction or a set of instructions some fixed number of times or while some condition is true.

**6. Subprograms (Subroutines) :** These are separately named sets of instructions that can be executed anywhere in the program just by a reference to the name.

The following is a brief discussion (using Pascal Language) of the last four elements :

**Input and output :**

Entering data into the computer and transfering data out of the computer are two essential activities in computer programming. These activities are called input and output activities.

In Pascal, the input activities are performed by two statements namely read and readin; and the output activities are performed by two statements namely write and writeln.

The read statement is used to read data from the keyboard (or an input device) and assign them to variables.

**Example :** read (x, y, z);

The write statement is used to print the data on the screen (or any output device)

**Example :** write ('An example of Pascal program');

The Pascal program to find the area and perimeter of circle is given below :

```
program circle (input, output);
const
        pie = 3.14;
var
        radius real;
BEGIN
        readd (radius);
        write ('Are of circle is', pie *radius* radius, 'square metres');
        write ('Perimeter of circle is', 2.0 *pie* radius, 'metres');
END.
```

This program accepts any real value for radius and prints the area and perimeter of circle accordingly.

**Conditional Execution :**

Generally, the statements in a program are performed in a sequential manner. However, in some programs we need to alter the sequence of steps depending on whether a condition is true or false. The conditional execution statement include a condition which is called a Boolean expression.

IF-THEN statement

The format of the statement is

```
if Boolean expression then
        statement;
```

If the Boolean expression is true then statement is executed. Otherwise, statement is not executed and control is transferred to the next statement after the if statement.

The execution of IF-THEN statement can be represented as

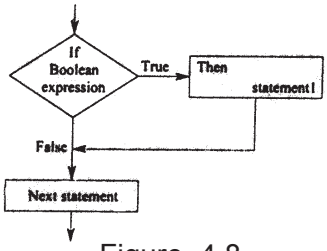



Figure 4.8

The following Pascal program read two numbers and finds the small number, big number and the difference of two numbers.

```
program difference (input, output);
        var
                x, y, temp : real;
                diff : real;
        begin
                write ('Enter two numbers:');
                readln(x,y);
                if (x>y) then
                begin
                        (interchange the value of x and)
                        temp : = x;
                        x : = y;
                        y: = temp
                end;
                diff : = y - x;
                writeln ("The small number is', x);
                writeln ('The big number is', y);
                writeln('The difference between two numbers is', diff)
        end.
```

IF-THEN-ELSE statement

The format of the statement is

```
if boolean expression then
        statement1
else
        statement2;
```

If the boolean expression is true then statement1 is executed. If the boolean expression is false then the statement2 is executed.

The execution of IF-THEN-ELSE statement can be represented diagramatically as

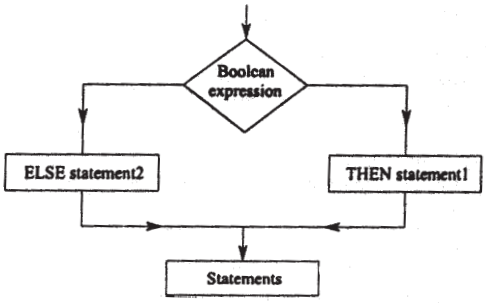



Fig. 4.9:

The following Pascal program check whether a given number is even or odd.

```
program even odd (input, output);
{This program reads a integer value and check whether the
number is even or odd
var
        N : integer;
begin
        write ('Enter value for N:');
        readln (N);
        if (N div 2) * 2 = N then
          writeln('Thenumber', N, 'is even');
        else
          writeln('Thenumber', N, 'is odd');
 end.
```

CASE Statement

IF-THEN-ELSE statement enables a programmer to choose a statement between two alternative statements. The CASE statement allows a programmer to select one statement out of many alternatives.

General format of CASE statement is

```
Case expression of
        Value1: statement1;
        Value2: statement2;
                :       :
                :       :
        Value n : statementn;
End;
```

where *experssion* is *case* selector or *case index* and it must have a value of ordinal type.

The value1, value2,...., statement *n* are *labels,* referred to as casse *labels;* they must be of the same type as the case selector.

Statement1, statement2,....., statement *n* are simple or compount statements.

*The case labels are different from the labels of statements defined in Pascal.* Case labels may be integers, characters, boolean or enumerated data items.

For example,

```
Case option of
        1: writeln('Selected first option');
        2: writeln('Selected second option');
        3: writeln('Selected third option');
        4: writeln('Selected fourth option');
end;
```

In this example, depending on the vaule of option the corresponding labeled statement is executed.

The execution of CASE statement can be represented as :

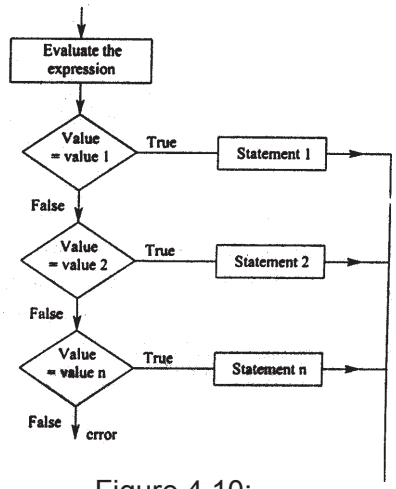



Figure 4.10:

The case statement operates as follows. First values of the *case expression*, that is, the *selector*, is obtained adn compared with each of the *case labels.* The statement whose label matches with the selector is executed and then control goes to the statement innediately following the case *end.*

The follwoing Pascal program reads mothe and year and prints the number of dyas in that month.

```
Program noofdays (input, output);
var
        totdays : integer;
        month, year : integer;
begin
        write ('Enter the month (mm) and year (yyyy);');
        readln (month, year);
        if      (month < 1 or month > 12) then
                writeln ('Not a valid month')
        else
           begin
                case month of
                1,3,5,7,8,10,12         : totdays :=31;
                4,6,9,11                : totdays :=30;
                2                       : if (year mod 4 = 0) and
                                          (year mod 100<>0) or
                                          (year mod 400=0)
                                                totdays : =29;
```

```
                                        else
                                                totdays : =28;
                end;
                        writeln ('The total no. of days is', totdays)
        end.
```

*Loop statements :*

Serveral time we have to repeat a statement or a sequence of statements a number of times or until some condition is satisfied.

Pascal provides three languag3es constructs for repetitions of statements.

i. FOR - DO statement
ii. WHILE - DO statement
iii. REPEAT - UNTIL statement.

**i. FOR - DO statement :**

FOR - DO statement is used when a sequence of statements are to be repeated for a known numberr of times.

*The FOR structure has two different forms*

1. **For** control_Variable:=initial_value **TO** final_value **DO** statement;
2. **For** control_Variable:=initial_value **DOWNTO** final_value **DO** statement;

The execution FOR-TO-DO can be represented as

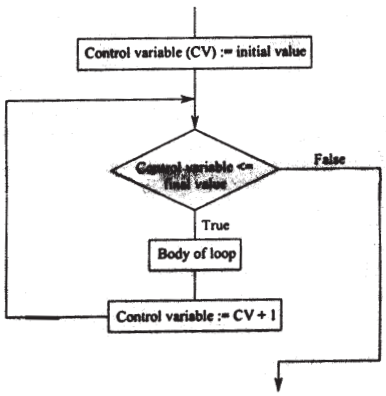



Figure 4.11:

The **control_variable** can be of any ordinal datatype (i.e., integer, char, booleadn, enumerated or subrangre). It should not be real datatype.

In the first form of **FOR-DO** statement, the *control_variable* automatically takes successive values (increment by 1 in case of integer type) starting form *initial_value* to *final_value* in the **ascending** order.

In the second form of **FOR-DO** statement, the *control_variable* automatically takes successive values (increment by 1 in case of integer type) starting form *initial_value* to *final_value* in the **descending** order.

Initially, the *control_variable* is assigned to value specified by *initial_value.*

For each repetition, the statement will be executed and the *control_variables* take successive values. Then, the value of *control_valriable* will be compared with *final_value.* This repetition continues until it reaches the *final_value.* After that the control goes to the next statement innediately after **FOR** statement.

Hence, the statement (loop body) will be executed.

(final_value-initial_value + 1) times.

*For example,*

```
for i:= 1 to 10 do
    read (a);
```

When this **FOR-DO** statement is executed, the value of variable a will be read 10 times. Here, i is a control variable which takes values from 1 to 10.

The values of initial value and final values of **FOR-DO** loop can be constants, variables or expressions. These values must ve of the same datatype as the control variable. The initial values and final value are evaluated once in the beginning and give initial and final values.

Consider the following **FOR-DOWNTO-DO** statement,

```
for i:= 10 downto 1 do
    read (a);
```

In the case also the values of a will be read 10 times.

But the control variable *i* takes the values 10,9,8,7,....2,1 for each repetetion.

ii. **WHILE-DO statement :**

**WHILE-DO** statement enables a programmer to execute a sequence of statements, a number of times so long as the sepcified condition is true.

General format of **WHILE-DO** statement is

**while** boolean expression **do**
statement>

The statement may be simple or compound statement.

The execution of **WHILE-DO** loop can be represented as

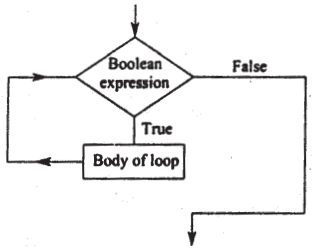



**Figure 4.12: EXECUTION OF WHILE-DO STATEMENT**

First, the value of the boolean expression is obtained. If the value is true, statement is executed and control goes back. Again the boolean expression is evaluated and its value tested. If the value is true, stateme -nt is executed, otherwise control goes to the statement is execution of the **WHILE-DO** statement is completed only when the boolean expressi -on is found to be false. Boolean expression is evaluated for each execution of statement.

*In the* **WHILE-DO** *loop, is the boolean expression is false initially, the statement will not be executed at all. If the boolean expression is true, statement may be executed one or more times.*

An example of while-do statement is

```
count :=1;
        while count < = 10 do
        begin
                read (a);
                count := count + 1
        end;
```

The statements between begin and end are executed as long as count is less than or equal to 10. When count becomes greater than 10, those statements will not be executed.

The following program segment reads and counts characters until the character read is a period (.).

**Example**

In an experiment, the readings are noted at regular intervals. The reading are added to previous total value at each interval. When the total exceeds a specified threshold then the program will display how many reading have been processed.

```
program experiment (input, output);
    const
        threshold = 200.0;
    var
        reading, total : real;
        interval : integer;
    begin
        interval :=0;
        total :=0;
        while (total <= threshold) do
            begin
                readln (reading);
                total := total + reading;
                interval := interval + 1;
            end;
        writeln ('The threshold exceeds at', interval,
```

```
                    'interval');
            end.
```

iii. **REPEAT - UNTIL** statement :

The format of statement is

```
                    Repeat
                        Statement1;
                        Statement2;
                        Statementn
                    Until boolean expression;
```

The execution of **REPEAT - UNTIL** can be represented as

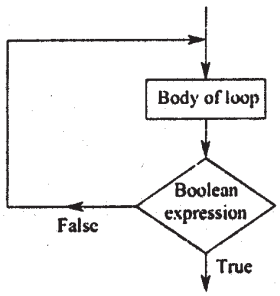



**FIGURE 4.13; EXECUTION OF REPEAT-UNTIL STATEMENT**

The statements are executed first and then the *boolean expression* is tested. If this value is false, the statements 1 to n are executed again. This goes until the value of the boolean expressionis true. Then control goes to the statement following the *until* keyword.

*For example :*

```
conunt:=0
repeat
        readln (C);
        count := count +1;
until (C='.');
```

This statement reads character and counts them until a period (.) is read.

The following Pascal program finds LCM (Lowest Common Multiple) of two numbers x1 and x2.

LCM of x1 and x2 can be written as

LCM = (x1*x2)/GCD of x1 and x2.

where GCD is Greatest common divisor of x1 and x2.

```
program lcm (input, output);
var
        x1, x2 : integer;
        lcm1, numerator, remainder, gcd : integer;
begin
        write('Enter values for x1 and x2');
```

```
            readln(x1, x2);
            numerator :;= x1* x2;
        repeat
            remainder:= x1 mod x2;
            x1:= x2;
            x2:= remiander
        until (x2 = 0);
        gcd: = x1;
        lcm := numerator div gcd;
        writeln('LCM of', x1 'and', x2, 'is' lcm1)
end.
```

### *Subprograms:*

When writing a program, we may like to repeat a statement or a group of statments may times. For example, we like to do certain compu tations many times. Pascal allows writing such type of programs in modular fashion, *by dividing the problem into independent sub problems.* These modules of program are called *subprograms.* Thus, a subprogram is *self-contained block of statments that performs a particular task.*

Subprograms may be combined to form larger programs. This process of breaking a big or complex program into simple subprograms to obtain the complete solution is referred to as modular design of programs.

Subprograms consist of one or more basic statements that can be referred at different points in a program. This will allow us to develop program easily by writing a simple, self-contained program units.

*The advantages of writing subprograms are :*

**1. *Provide modularity :***

Subprograms allow breaking a larger program down into manageable, well defined logical modules. This supports top-down design and step-wise refinement approach of problem solving.

**2. *Provide reusability :***

Subprograms can be used any number of times in a program. This eleminates the need of repeating the same code many times.

**3. *Aid readability :***

Subprogram increase the clarity of programs by separating the low-level details form the main program logic.

**4. *Provides sharability :***

Subprograms developed by others may ube used or even shared

through program libraries.

**5. *Promote maintainability :***

If the definition of subprogram changes, it will effect only the subprogram but not entire program.

There are two kinds of subprograms available in Pascal : Functions and Procedures. Generally a procedure is used to perform an action and a function is used to compute a value. The following is the brief description of functions Pascal language.

A function is an independent unit in Pascal program. It has its declarations and executable part. The execution part is also referred to as the body of the function subprogram.

Consider an example to compute factorial of N.

The factorial of N can be defined as

```
N! = 1                                          if N = 0
   = N * (N-1) * (N-2).....*2*1                 if N > 0
```

A function subprogram is invoked (or called) as part of a stateme - nt from main program, or any other, sub program within the scope (scope rules are discussed in the next section). The call is made by sepecifying the functionname along with paramaters list if any.

*For example,* the statement

```
x := fact (5);
```

in the main program, invokes the factorial function.

Function references are also known as function calls.

The function subprogram, which is invoked is called **called program** and the progaram from which the functions invoked is called **calling program.**

The following Pascal program find the factorial of first N numbers.

```
Program main program (input, output);
   {This program is an example for function subprogram}
var
        x,i,j : integer;
function factorial (n : integer);
var
        i, product: integer;
begin
        if n = 0 then
```

```
                    factorial := 1;
                else
                    begin
                        product := 1;
                        for i := 1 to n  do
                            product:= product* i;
                        factorial := product;
                    end;
end;
begin
    write('Enter the values for N:');
    readln(x);
    for i : = 1 to x do
    begin
        j := fact (i);
        writeln('The factorial of', i,'is',j);
    end;
end.
```

## CHAPTER – 3

# FEATURES OF 'C'

## 3.1. Introduction to 'C'

This chapter deals with the fundamental elements used to construct simple C statements. These basic elements include the C character set, keywords, identifiers, variables, constants, datatypes, declarations, statements and expressions.

The basic data objets, variables and constants, are manipulated in a program. Declarations list the variables to be used along with their type and perhaps their initial values. The type of the object determines the set or range of values it can have and what operations can be performed on it.

Operators specify what is to be done to the objects. Expressions combine variables and constants to produce new values.

### Character Set

Any language has its own building blocks. For example, in English we have the letters A through Z, which are used to construct words and sentences. C's building blocks are the character set. The character set consists of :

- uppercase letters A to Z
- lowercase letters a to z

**Note :** It is important to note that, C is case sensitive. Uppercase and lowercase letters are not interchangeable. For example, the words 'Tom' and 'tom' are treated differently.

- digits 0 to 9
- special character :

  + - * / = ( ) { } [ ] < > ' " ! @ # $ % & ... | ^ ~ \ . , ; : ? 'blank space'

**Whitespace :** A blank space or sequence of characters containing only spaces, tabs, newlines, and form feeds is called whitespace.

**Escape Sequences :** C also contains certain combinations of character sequences referred to as escape sequences. Each combination is considered to be one character. Table 2.1 lists the escape sequences and their purposes.

| Escape Sequence | Purpose |
|---|---|
| \" | double quotation mark |
| \' | single quotation mark |
| \\ | backslash |
| \? | questino mark |
| \a | alert (bell or beep) |
| \b | backspace |
| \f | form feed (new page) |
| \n | newline |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |
| \ddd | character represented by octal code |
| \xhhh | character represented by hexadecimal code |

Table 2.1 Escape Sequences

The following example uses the escape sequence \" to print "Paris" is the fashion city of the world.

```
printf ("\"Paris\" is the fashion city of the World");
```

These characters are collected by the compiler into syntactic units called tokens.

## Identifiers

Identifiers are tokens written as a sequence of letters, digits, and underscore(s). These are the names given to program elements such as vaiables, functions, arrays, etc. The following rules apply for constructing identifiers :

- An identifier may consist of letters (both upper and lower case), digits, and underscores ( _ ).
- An identifier must begin with a letter or an underscore, not with a digit.

Other points to keep in mind when constructing identifiers are :

- Upper and lower case letters are not interchangeable.
- Follow the naming conventions. Choose meaningful names as identifiers such as tax_rate, price, tax,etc. The common practice is to use lower case letters for variables.
- An underscore ( _ ) character is sometimes used to represent a space.
- Though, an identifier can begin with an underscore, avoid such naming because of possible conflicts with system identifier names. Such identifiers may be reserved.
- Spaces are not allowed in an identifier.
- ANSI C recognizes at least the first 31 characters of an identifier though some C compilers can recognize more.
- An identifier cannot be the same as a keyword, or an identifier already defined in the library, such as printf().

The following are some valid identifiers :

| | | | | |
|---|---|---|---|---|
| x | sum | area | PI | table |
| y | names | tax_rate | i_am_an_identifier | |

The following are some invalid identifiers and the reasons they are invalid:

| | |
|---|---|
| 4th_class | the first character must not be a digit |
| "x" | illegal character (") |
| order-no | illegal character(-) |
| tax rate | illegal character (blank spaces) |

**Keywords**

Certain identifiers have a predefined meaning and are reserved for a specific purpose. Such identifiers are referred to as keywords. Table 2.2 lists the 32 identifiers which are reserved for use as keywords. They cannot be used for any other purpose. We will discuss them in later chapters.

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

Table 2.2 Keywords

Remembering that C is case sensititive, you should note that keywords are lower case. Some C compilers may recognize other keywords such as fortra, asm, etc. You are encouraged to consult the reference manual to obtain a complete list of keywords.

**Variables**

A variable is a named memory location used to store values that may change. A variable is also called an identifier. All variables must be declared before they are used.

For example, let's say you want to use the redius of a circle in a program for calculation. You may want to start with a circle of one size, but be able to change the size for later program runs. To accommodate this, you would want to use a variable. You may want to name the variable 'radius'. Now, based upon the radius, the area of the circle changes. Therefore, you also want to use a variable to store the value of the area. You may want to name it 'area'.

Now you have two variables, radius and area, which should be declared in your program. Based upon the declaration, the appropriate amount of memory is reserved for each of these variable. Each of the memory locations can be addressed by the variable's name.

When you give radius a value, say 10, this value is stored in the memory location reserved for radius. Let us say you also have a statement in your program to compute the area. Once the area is computed, it is stored in the memory location reserved for area. When you give a printf() statement to display the area, this value will be fetched from the memory location of area.

To put it in simple terms, you can imagine a variable to be like an empty container which can hold anything. For example, a container called water_cup to hold water, coffee_cup to hold coffee and wine_glass to hold wine, etc. Here water_cup, coffee_cup, and wine_glass are all variables. Further, a water cup can hold any water like mineral water, salt water, or pure water. Similarly, if radius is a variable to hold whole numbers (which are referred to as decimal integers in C language), it can hold '10' or '12' or '4' or any decimal integer.

Depending upon the size, a water cup can hold certain maximum amount of water, and so does a variable. If you have a water cup that holds 25 ml. it can only hold a maximum of 25 ml of water. If you need to hold more water, you need a bigger holding container. A variable works in much the same way. To store a bigger number. If you need to a variable type which can provide more storage. Therefore, the type of variable you use will

depend upon the size and type of data that you want to store.

## Datatypes

C language offers four fundamental types of variables : char, int, float and double.

**Char :** a variable type that holds a single byte (8 bits). It is capable of holding one character.

**int :** an integer type capable of holding whole numbers The range of values that can be stored in this variable type is machine dependent. For example, the range for a 16 bit integer is from -32768 to 32767.

**float :** a single precision floating point type for storing real numbers.

**double :** a double precision floating point type for storing real numbers. This provides higher acuracy than the float type.

In addition, there are a number of qualifiers that can be applied to the basic types. Qualifiers like short and long can be applied to integers to provide for different lengths of integers. The qualifiers signed or unsigned can be applied to char or int Unsigned numbers are always positive or zero. Table 2.3 lists the common values for these types.

| Type | Bits Used | Range of Numbers |
|---|---|---|
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | -128 to 127 |
| int | 16 | -32768 to 32676 |
| short int | 16 | -32768 to 32767 |
| long int | 32 | -2,147,483,648 to 2,147,483,647 |
| unsingned int | 16 | 0 to 65,535 |
| unsinged short | 16 | 0 to 65,535 |
| unsigned long | 32 | 0 to 4,294,967,295 |

Table 2.3 Datatype Values

These values are dependent on the machine and compiler. The standard header files <limits.h> and <float.h> contain the constants for all these types, along with other properties for the machine and compiler.

It is the programmers responsibility to select the appropriate datatype for each of the variables. You have to make sure that the data is within the range of the variable, otherwise you will get erroneous results, but not an

error message. For example, say you have int num1, num2, and sum. If num1-30000 and num2=3000, then sum, which is of decimal integer type, will be equal to 33000. This is beyond the range of a decimal integer (-32768 to 32767) and will give you incorrect results.

### 1.4.2 Declaration of Variables

In programming, when you want to manipulate certain data, you can write/store it to a specified memory location, and address it or fetch it as needed. That is, before using a variable, you should have a specified storage location for this variable and a means to address it. This is achieved by variable declaration. The vaiable declarationdefines the variable's type and sets aside storage for the variable. All variables must be declared before they can be used. A common practice is to declare variables at the start of a function block before any executable statements. Variables can also be declared in other blocks (while, for, etc.) and such variables are local to the block.

```
Syntax      type variable1, variable2, variable3;
            or
            type variable 1;
            type variable 2;
            type variable 3;
```

Examples of this are as follows :

```
int number1, number2, number 3;
        or
int number 1;
int number 2;
int number 3;
```

The following statement sets aside the memory required for a decimal integer (two bytes). This storage can be addressed by the name 'radius'.

```
int radius :
```

The following statement sets aside the memory required fora variable of the float type (four bytes). This storage can be addressed by the name 'area'.

```
float area;
```

### 2.4 Contants

As opposed to variables, the values of a constant will not change. For example, the value of '5', 'P' or 456.72 remains the same. As the name

implies, the value will always be constant.

### 1.5.1 Notation

An integer constant like 3456 is an int. A long constant is written with a terminal 'l' or 'L', as in 987654321L. An integer too big to fit into an int must be taken as a long contant. Unsigned constants should be written with a terminal 'u' or 'U'. The suffix 'ul' or 'UL' indicates an unsigned long constant. Floating point constants contain a decimal point (234.7), an exponent (1e-2), or both. Their type is double, unless suffixed. The suffixes 'f' or 'F' indicate a float constant, 'l' or 'L' indicate a long double constant.

The value of an integer can also be specified in octal instead of decimal. A leading zero (0) on an integer constant specifiesoctal. For example, the decimal number 125 can be written as 000175 in octal notation.

The value of an integer can be specified in hexadecimal instead of decimal. A leading 0 x or 0X (zero followed by 'x' or 'X') means hexadecimal. For example, the decimal number 125 can be written as 0x7d or 0X7D in hexadecimal notation.

### 1.5.2 Character Constants and Constant Expressions

A negative constant integer such as - 25 is considered a constant expression. The character contant is a special integer, written as one character within single quotes, such as 'x', 'Z' or '+'. Each character has a particular integer value which is dependent upon the character set used on a particular computer.

### 1.5.3 String Constants

A string constant is a sequence of characters enclosed in a pair of double quotation marks, such as "xyz". Another name for a string constant is a string literal. Note that the double quote is a single character, not two characters. Also, "a" and 'a' are not the same. The following are a few example of string constants :

```
"a string of test"
" "                                          /*this is a null string*/
"          "                        /*this is a string of blank characaters*/
"a=b+c;"              /*this is string constant. Nothing is executed*/
" /*this is not a comment*/"
"a string with a single backslash \\within"
"a string with a double quotes\"within"
```

An example of an invalid string is : /* "this is an invalid string" */. |* is invalid

because the double quotes are enclosed within the /* */ which designates a comment.

### 1.5.4 Symbolic Constants

In C you can associate a symbolic name with a contant. You can take a constant such as IT which is equal to 3.142857 and make it a symbolic constant named PI.

A symbolic constant is a name given to a sequence of characters. The characters may represent a numeric constant, a character constant, or a string constant. In the above example, PI is the symbolic contant used to represent the numeric (float) constant 3.142857. During precompilation each occurrence of the symbolic constant is replaced by its value. Symbolic constant names are conventionally written in upper case so that they can be readily distinguished from lower case variable names.

### 1.5.5 Enumeration Constants

In some other cases, you may have a special type of constant which can only take a value from a given set. For example a day can be one of the seven days in a week, and similarly a month can take any value of one of the twelve months of a year. Another example could be a symbolic constant choice which can take either no or yes. For such situations C provides a special type of constants called enumeration constants.

An enumeration constant is a list of contant integer values. For example, you want to declare a symbolic constant called boolean which can take one of two values : NO or YES. The syntax for this is shown in the following example, where enum is the type, boolean is the enum variable and NO and YES are the enumeration constants with values of 0 and 1 respectively.

```
enum boolean (NO, YES);
```

The first name in an enumerated constant has a value of 0, the next a value of 1 and so on, unless explicit values are specified. In the following example, JAN has a starting value of 1, so FEB will be 2 and MAR will be 3, and so on.

```
enum months {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
                    SEP, OCT, NOV, DEC};
```

Names in different enumeration constants in the same program should be distinct, as they are actually not associated with the enum variable. The names are used as constants directly.

### 1.5.6 Declaration of Contants

Contants are usually declared using the #define preprocessor directive. A #define line defines a symbolic name or a symbolic constant to be a particular string of characters :

| Syntax | #define name replacement text |
|---|---|

Thereafter, any occurrence of the name will be replaced by the corresponding replacement text during preprocessing. In the following example, PI is declared as a constant which is equal to 3.142857

```
#define PI 3.142857
```

Note that symbolic constant names are conventionally written in upper case so that they can be readily distinguished from lower case variable names. Also note that there is no semicolon at the end of the #define line.

Preprocessing takes place just before compilation. During this process, the occurrence of the symbolic constant is replaced by its value (except for those in string constants). Thus in the following example, the preprocessor changes :

```
printf("PI=%f", PI);
```

to

```
printf("PI=%f", 3.142857);
```

### 1.5.7 Const

ANSIC also provides a type qualifier called const, which can be used to declare a constant.

| Syntax | const int n = 3; |
|---|---|

Since n has been qualified by a const, we can only initialize n. Thereafter, n cannot be assigned to, incremented, or decremented. In other words, once initialized its value cannot be changed.

In a declaration, const comes before the type. A const qualified variable is not eaequivalent to a symbolic constant (constant defined by the #define), because, #define is a preprocessor directive and doesn't take up any memory, whereas cont reserves the memory appropriate to the datatype.

## 1.6 Statements

A statement causes the computer to perform a specific action. C has three

different classes of statements; expression statements, compound statements, and control statements.

### 1.6.1 Expression Statements

An expression statement consists of an expression followed by a semicolon. When this statement is executed, the expression gets evaluated. An assignment statement is a type of expression statement.

### Assignment Statements

How do variables get their values? You want the variable radius to hold a value of 10. You've already created an addressable memory location for radius when you declared the variable. Now, how do you put the value 10 into the memory location reserved for radius? There are several ways of doing this. The most commonly used way is by an assignment statement. Assignment statements can be written along with the declaration statements as shown in the following example. In this example, a, b and c are variables of the type integer. 0 (zero) is stored in a , 10 is stored in c, and b is declared as an integer variable, but it has no value stored in it.

```
int a = 0, b, c = 10;
```

In the following example, letter is a variable of type char and holds the character D. Similarly, the character '-' is stored in the variable named sign. Symbol is also a variable of type char, but has no value stored in it.

```
char letter = 'D', sign = '-', symbol;
```

There are several types of assignment statements including simple assignment statements, compound assignment statements and nested assignment statements.

#### Simple Assignment Statements

```
Syntax    expr1 = expr2 op expr3;
```

Here, expr2 and expr3 are the two operands for the operator op. The result of expr2 op expr3 is calculated first and is then stored in expr1. In other words, the expression on the right-hand side is evaluated first and the final result is stored in the variable present on the left hand side of = (the assignment operator).

In the following example, b is added to c, and the result is stored in a. The values of c and b are not changed whereas the value of a is equal to the sum after the compiler executes the statement. The operator + acts on the two operands b and c.

```
a = b + c;
```

The operators + - * / % are referred to as binary operators because they act on two operands.

Compound Assignment Statements

If we have a = a + b; we can write the statement as:

a+ = b; which means a is added to b, and the result is stored in a.

```
Syntax   expr 1 op = expr2;
```

expr1 stands for expression1, op for operator, and expr2 for expression2. expr1 and expr2 are operated on by the operator op, and the result is stored in expr1.

In the following example, a is added to b and the sum is stored in a. This is the same as a = a + b;.

```
a + = b;
```

In the following example, a is divided by c and the result is stored in a. This is the same as a = a / c;

```
a / = c;
```

In the following example, a is divided by d and the remainder is stored in a. This is same as a = a % d;.

```
a = % d;
```

Nested Assignment Statements :-

```
Syntax    expr1 = expr2  = expr3 = expr4;
```

Theresult of expr4 is calculated and stored in the variables expr3, which in turn is assigned to expr2, which in turn is assigned to expr1. So expr1= = expr2 = = expr3. Please note that = is an assignment operator whereas == is equal to.

In the following eample, the nested statement's result (give on the left) is the same as if you executed the individual statements (given on the right) -- a value of 10 is stored in a, b & c.

```
a = b = c = 6+4;          is the same as      a = 6 + 4;
                                              b = 6 + 4;
                                              c = 6 + 4;
```

### 1.6.2 Compound Statement

A compound statement consists of several individual statements enclosed within a pair of braces {}. The individual statements can be either expression statements, compound statements, and/or control statements. Note that a compound statement does not end with a semicolon.

### 1.6.3 Control Statements

A control statement is used for performing special program features such as logical tests, loops, and branches. Based upon a specified condition(s) in the logical test(s), or loops, the control is shifted. Many control statements need other statement(s) embedded within them. Control statements are discussed in depth in Chapter 5.

## 1.7 Expressions

A single data item such as a number or a character is an expression. For example, 'b' is an expression. The simplest expressions are contants, such as 5 or 29. The name of a variable alone can be considered an expression. Any meaningful combinations of operators with variables and contants is also an expression. For example, b + c is an expression.

For example, in the statement a = b;, a is an expression, b is an expression and c is an expression. Further, b + c is an expression. Each expression has its own value. The above statement implies that the value of the expression b + c is evaluated and assigned to expression a. Furthermore, a = b + c is also an expression.

Consider b = 3 and c = 5. 3 is an expression and has the value 3, which is assigned to expression b. b = 3 is an expression which has a value 3. Similarly, 5 is an expression which has the value 5, which is assigned to c and hence the expression c = 5 gets this value.

In the expression a = b + c, the expression a gets the value 8, and the whole expression a = b + c also has the value 8. Note that a semicolon placed at the end would have made this an assignment statement (a = b + c; is a statement).

Now, consider the statement; a = a / b; a is divided by b and the result is stored in a, which is on the left hand side. The b value is not changed, but the value of a is changed. This is equal to the quotient (a/b).

Expressions can also represent a logical condition such as true or false. It is important to note that in C, the logical conditions true and false are represented by the integer values 1 (any non-zero value) and 0 (zero).

## Chapter 3 - Exercises

1. What characterscomprise the C character set?
2. What is an identifier and why do you need identifiers? What are the rules for naming identifiers?
3. Give five examples of valid identifiers. Give five unique examples of invalid identifiers along with the reason(s) they are invalid.
4. What are variables and constants?Give four examples for each.
5. What is a keyword? Name some keywords in C.
6. Is the decimal integer 1 the same as binary 1? Explain.
7. Explain the different types of constants, give examples.
8. How do you distinguish between integer constants, octal contants, and hexadecimal constants?
9. Distinguish between unsigned integer constants, long integer constants long integer constants, give examples.
10. Explain the different ways of expressing a floating point constant, give examples.
11. What is a character constant and how does it differ from a numeric constant?
12. What is a string constant? Give two examples.
13. Describe enumerated data type give two examples. How are they declared.
14. What is a symbolic constant? How are they declared? Explain with examples.
15. Explain the #define statement, give an example. How does it work?
16. Explain the use of a const type qualifier, give examples. What is the difference in using const verses #define.
17. What is an escape sequence? What is its purpose? Summarize the standardescape sequence in C.
18. Can escape sequences be included in a string constant? Explain with example(s).
19. Name and describe four basic datatypes in C.
20. Give the upper and lower limitsfor each type. What happens when the limits ae crossed for a particular datatype, for example, what happens when you try to store 40000 in an int type variable. How

should you handle such situations?

21. Do you have to declare all the variables that you use in the C program? What is the purpose of variable type declaration? Explain the different components of type declaration with examples.

22. What is an operator and what is an operand? Explain with examles.

23. What is an expression? Explain the differece between an expression and a statement, give examples.

24. What is an assignment statement? What are different types of assignment statements? Explain with examples.

25. What is a compound statement? Explain with an example.

26. Explain the nested assignment statement, give an example. What is the difference between a compound assignment statement and a nested assignment statement.

## CHAPTER – 4

# 1. INPUT / OUTPUT FUNCTIONS

The stdio (standard input/output line) library of C provides a number of input and output functions. The most commonly used functions are scanf(), getchar(), and gets() for input from a standard input device, usually the keyboard. The functions printf(), utchar(), and puts(), are commonly used for output to a standard output device, usually the screen.

The input/output (I/O) functions scanf(), and printf() are used for the input/output (respectively) of characters, character strings, and/or numbers. These two functions are more complicated then the other four input/output functions discussed in this chapter.

The functions getchar(), and putchar() are used for the input/output of character data (only one character). The functions gets() and puts() are used for the input/output of character strings (combinations of more than one character, such as name, address etc.). Once you know how to use these functions, you will be able to writ simple C programs.

## 1.1 Printf()

The printf() function outputs characters, character strings and numbers. The output format is specified as a contant string in double quotes.

```
Syntax    int printf(format, arg1, arg2,.....);
```

In the first chapter, we used a simple printf() statement to print the character string "Learning C is fun!"

```
printf("Learning C is fun!!");
```

In this example, the printf() function contains only one argument. The arguments for a function go within the parenthesis, in this case the character string "Learning C is fun!"

A printf() function can have more than one argument the first argument is a format string. The format specification string can contain the character string that you want to print, along with the conversion specifications describing how other values are to be converted into printed form. The values to be converted are supplied as additional arguments to the printf() function. Arguments are separated by commas.

When the format string is printed, each of the conversion specifications is replaced by the converted value. A conversion specification always begins

with a % sign. Table 3.1 lists the commonly used conversion specifications.

| Character | Represents |
|---|---|
| c | single character |
| d | decimal integer |
| e, E | double or float with an exponent representation |
| f | double or float with decimal notation (not exponent) |
| g | double or float of e-type or f-type (depending on the value, trailing zeros or a decimal point will be displayed) |
| i | signed decimal integer |
| o | octal integer (without a leading zero) |
| p | printing as a pointer (address notation) |
| s | character string |
| u | unsigned decimal integer |
| x | hexadecimal integer (without a leading 0x) |
| % | no argument is converted, print a% |

Table 3.1 Conversion Specifications

The following example uses %d to put the value of sum (50) into the character string. "The sum of the numbers is 50" will be output.

```
printf("The sum of the numbers is%d", sum);
```

The following example uses%d to put the values 20, 30 and 50 into the character string. "20 plus 30 equals 50" will be the output, and the escape sequence \n will cause the cursor to go to a next line.

```
printf("%d plus%d equals %d\n", 20, 30, sum);
```

In the following example, if float x = 2.5, y = 3.5 and product = x * y, the output will be "The product is 8.750000".

```
printf("The product is %f", product);
```

Using the same variable values, the following example will output "The product is 8.750000". Inthis case, you do not need a separate variable 'product'. Also, a %f conversion specification prints a floating point number with six decimal plaaces unless you specify otherwise.

```
printf(The product is %f', x * y);
```

In the following example, the statement will display the value of x as an integer.

```
printf("%d", x);
```

In the following example, the values of x and y are displayed as integers separated by a space.

```
printf("%d%d", x,y);
```

In the following example, the values of x and y are printed together as specified in the control string. Hence the control string actually controls how the data is displayed/output.

```
prinft("%d%d", x,y);
```

In the following example, x and z are displayed as integers and y is displayed as a float (real) number.

```
printf("%d%f%d", x,y,z);
```

In the following example, the value of the variable a is displayed as a character. Remember that a format conversion character is specified afer a % sign.

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

If the character after the % is not a conversion character, the behavior is undefined, prinft() returns the number of characters successfully written, or a negative numberif an error occurred.

Using the C program elements we have already discussed, the following is a simple program to compute the area of a circle, given radius = 10 units. This program will display the radius and the area of the circle.

```
/* ..................................................................................... */
/*  Name    : circle.c - Given radius = 10, this program computes and displays the area  */
/*              of the circle                                                            */
/*  Date    : 5/29/97                                                                    */
/*  Author  : Prasad Putta                                                               /*
/* ..................................................................................... */
#include <stdio.h>                          /* This is the standard input/output file*/

void main()                                 /* main functin is void type, meaning that
                                                  it will not return any data item but will
                                            only execute the statements in the block.*/
{
        int radius;
        float area;
        radius = 10;
        area = 3.142857 * radius * radius;
        printf("The radius of the circle is%d, and its area is %f', radius, area);
```

```
                /*will print: The radius of the circle is 10, and its area is 31.428574*/
    return;                         /*returns the control to the operating system*/
}
```

Sometimes you may want to control the width and/or justification of the output. As shown in Table 3.2, there are a number of formats which you can apply with a field width specifier. Table 3.3 lists the justification specifiers.

| Format | How it displays. |
|---|---|
| %d | print as decimal integer |
| %5d | print as decimal integer, at least 5 characters wide. |
| %f | print as floating point. |
| %5f | print as floating point, at least 5 characters wide. |
| %3f | print as floating point, 3 characters after decimal point. |
| %5.2f | print as floating point, at least 5 wide and 2 after decimal point |
| %o | print as octal number |
| %x | print as hexadecimal number |
| %c | print as character. |
| %s | print as character string |
| %hd | print as short integer |
| %id | print as long integer. |

Table 3.2 Field Width Specifiers

| Flag | Meaning |
|---|---|
| - | data itemis left-justified (blank spaces if any are added after the data item) |
| + | to have a preceding sign (+ or -). Without which only the negative items will have the preceding sign. |
| 0 | leading zeros will appear instead of leading blanks for a right-justified data item. |

Table 3.3 Justification Specifiers

In the following example, the value printed has a field width of 6 which means that the value 25 is right justified in a six-character field with four spaces in front of it. ('.' represents the spaces.)

| | |
|---|---|
| Input | printf("%6d",25); |
| Output | ....25 |

A minus sign preceding the field width causes the printed value to be left justified. In the following example, the value printed has a field width of 6

which means that the value 25 is left justified in a six character field with four spaces after it. ('.' represents the spaces.)

| | |
|---|---|
| Input | printf("%-6d",25); |
| Output | 25..... |

The field width can be followed by a decimal point and a precision specified. For a %f conversion, precision specifies the number of decimal places to be printed. The following example prints 5.6500 right justified in a eight character wide field.

| | |
|---|---|
| Input | printif("%8.41",5.65); |
| Output | ...5.6500 |

You can specify precisionalone, omitting the field width, as in the following example which prints 5.6500 in a six character wide field without any padding.

| | |
|---|---|
| Input | printf("%4f',5.65); |
| Output | 5.6500 |

Precision can also be specified for a %d conversion representing the minimum number of digits to be printed. In the following example, leading zeros will be printed to make up the specified precision.

| | |
|---|---|
| Input | printf("%.5d", 25); |
| Output | 00025 |

In case of a %s specification, precision specifies the maximum number of characters to be printed. In the following example, the string contains more than the specified number of characters and the string printed will be truncated on the right.

| | |
|---|---|
| Input | printf("%.5s","Programming"); |
| Output | Progr |

The following program demonstrate format specifications and procession.

```
Program /* ..................................................................... */
        /* Name   : printing, c- To demonstrate the format specifications and  */
                    precision.                                                 */
         /* Date   : 5/29/97                                                   */
         /* Author: Prasad Putta                                               */
        /* ..................................................................... */
```

```
#include <stdio.h>
void main()
{
        int x = 12345;
        float f = 345.678;
        doube p=5000.00  q=0.0025;

        printf("%f %f %f\n\n",p, q,p*q, p/q);
        printf("%e %e %e %e\n\n", p,q,p*q,p/q);
        printf("%3d %5d %8d\n\n", x, x, x);
        printf("%3f %10f %13f\n\n", f, f, f);
        printif("%3e %13e %16e\n\n", f, f, f);
        printif("%3g %13g %16g\n\n", f, f, f);
        printif(%7f %7.310f %7.5f\n\n", f, f, f);
        printf("%f %.3f %1f\n\n", f, f, f);
        printf(%12e %12.5e %12.3e\n\n", f, f, f);
        print("%e %.5e %.3e\n\n", f, f, f);

        return
}
```

Output

```
5000.000000 0.002500  12.500000  2000000.000000
5.000000e+003  2.500000e-003  1.250000e+001  2.000000e+006
12345   12345   12345
/*even if the field width is small the entie integer is displayed*/
345.678000 345.678000   345.678000 /*same comment as above*/
3.456780e+002  3.456780e+002  3.456780e+002
345.678       345.678       345.678
345.678000  345.678    345.7
345.678000  345.678    345.7
3456780e+002  3.45678e+002   3.456e+002
3.456780e+002  3.45678e+002  3.456e+002
```

## 1.2 Scanf()

Scanf() is the standard input function used in C. As with printf(), this function is defined in the stdio library. Scanf() reads input from the standard input stream stdin, usually the keyboard, and writes it to the address of the specified variable. Scan() produces formatted output.

As in printf(), parameters are passed into the scanf() function. The first parameter is the format string. The format string provides the conversion specifications, i.e., it specifies the type to which the input value is to be converted. The format string is followed by the address of the variable(s) where each of these values is to be stored.

| Syntax | int scanf(format, arg 1, arg2,.....); |
|---|---|

In the following example, the scanf() function contains two parameters. The first parameter which is the format string specifies that the data value should be read as a decimal integer. The second parameter & x (read as ampersand x), specifies that the input value is to be stored in the address location o the variable x. Once this statement is executed, the input data item read is converted into decimal integer type and stored in the address of the variable x.

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

In C, addresses are referenced with the address of operator(&). If x is a variable, then &x is the address of x. This address is a value and can be passed to a function just like any other parameter or argument (as shown in the above example).

It is important to note that x and &x are different. x is the name of the variable, whereas &x is the address or the memory location where the value of x is stored. Therefore, the statement scanf("%d", x); will give unpredictable results. Usually the program will crash and/or give you an error message.

The only exception to this is when you are reading input of the type array such as character strings. In the following example, where char name (20); name is a character array which can hold a maximum of 20 characters. In this case, the scanf() function will be :

```
scanf("%s", name);
```

The format conversion specification characters for scanf() are almost the same as for printf(). There are a few exceptions like the conversion specifier for the double data type, which is 'f' for output but 'if' for input. Table 3.4 lists some commonly used format conversion characters.

| Character | Used to represent |
|---|---|
| c | single character |
| d | decimal integer |
| e | double with exponential representation |
| f,g | real values stored in a float variable |
| i | integer (leading zero for octal or leading 0x or 0X for hexadecimal) |
| o | octal integer (without leading zero) |
| p | reads a pointer (address notation) as printe by printf. |

| | |
|---|---|
| s | reads a character string |
| u | unsigned decimal integer |
| x | hexadecimal integer (with or without leading 0x or 0X) |
| % | literal %, no assignment is made |

Table 3.4 scanf() Conversion Characters

There are also length modifiers for scanf(). Table 3.5 lists these.

| Character | Description |
|---|---|
| h | preceding d to read in a short integer |
| i | preceding d to read in a long integer |
| l | preceding f to read in a double. |

Table 3.5 scanf() Length Modifiers

For example :

```
printf("Please enter the length of the field/n");
scanf("%lf", & length);
printf("The length of the field is %f\n", length);
```

In the following example, an integer value is read into the address of an integer variable x.

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

In the following example, there are three input items each separated by a whitespace. The first data item is read in as a decimal integer value and is stored in the address of the variable x. The second data item is read in as a float type and is stored in the addess of the variable y. The third value is read in as a decimal integer data type and is stored in the address of the variable z.

```
scanf("%d %f%d", &x, &y, &z);
```

If you have the three values separated by commas as in the following example, then the scanf() statement will read as:

```
scanf("%d, %f, %d", &x, &y, &z);
```

In the following example, a value is read in as a character into the address of the character variable a.

```
scanf("%c", &a);
```

scanf() returns end of file (EOF) if the end of file character is found or, the number of input items successfully read.

The following is a simple program to compute the area and perimeter of a rectangle. This program prompts the user for length and width, reads them in, computes the area and perimeter then displays the results.

Program

```
/* ..................................................................... */
/* Name : rectang.c - Interactive program to compute & display area & */
/*          perimeter of a rectangle                                    */
/*  Date   : 5/29/97                                                    */
/*  Author : Prasad Putta                                               */
/* ..................................................................... */
#inlude <stdio.h>       /* This is the standard input/output header file*/
int main()              /* main functin is int type, meaning that
                           it will return a decimal integer once all
                           the statements ae executed sucessfully */
{
    double length, width;                                   /* data */
    double area, perimeter;                               /* results */
    printf("Please enter the length of the rectangle: ");
                                        /* prompts the user for the length*/
    scanf("%if", &length);
   /*scans the value as double and stores it at the address of length*/
    printf("Please enter the width of the rectangle: ");
    scanf("%1f", & width);
     /*scans the value as double and stores it at the address of width*
    area = length * width;
    perimeter = 2 * (length + width);
    printf("The area of the rectangle = %.3f\n", area);
                          /*will print the area according to the specified
                       precision and the cursor to move to the new line*/
    printf("The perimeter of the rectangle = %.3f/n", perimeter);
    return 1;                       /*returns 1 to the operating system*/
}
```

Output

```
Please enter the length of the rectangle : 7.25      /*user enters 7.25*/
Please enter the width of the rectangle : 4.9         /*user enters 4.9*/
The area of the rectangle = 35.525
The perimeter of the rectangle = 24.300
```

### 1.3 Advanced scanf() Features

Beginning C programmers may want to skip this section. Some advanced features of scanf() are discussed.

You can have precision specified in the control string of the scanf() function. In the following example, if the input is 1,3, and 5, then the execution of the scanf() statement will result in a = 1, b = 3, and c = 5.

```
int  x, y, z;
scanf("%3d %3d %3d", &x, &y, &z);
```

On the other hand, if the input is 123,  456, and 789; or, 123456789, the execution of the scanf() statement will result in a=123, b=456, and c = 789.

Using the same scanf() statement as inthe above example, suppose the input is 1234, 56 and 789. The result will be a = 123, b = 4, and c = 56. Using this data, the number 789 is ignored. It is ignored because the precision of reading variable a is 3. Only the first three digits are read into a. The next alue read is for the variable b whose precision is also 3. After reading 4, a whitespace is encountered which indicates that the data item ended there. Therefore the variable b gets the value 4. The next variable read is c whose precision is also specified as 3, but here the next value is 56 followed by a whitespace indicating the end of the data item. Therefore the value 56 is read into the variable c.

Since scanf() starts to read only the three variables which are already read, the rest are ignored, hence the number 789 is left out.

In the following example, if the input is 22, 356.85 and C, then the execution will result in; x = 22, Y = 356.8 and z = 5.

```
int x; float y; char z;
scanf("%3d %5f %c", &x, &y, &z);
```

In the following example, if the input is 5, a, and 3.9,  then the execution of the scanf() causes the first value to be rea dand assign to the variable x. The second value, a, is read and ignored. The third value, 3.9, is read and assigned to y.

```
int x; float y;
scanf("%d a %f", &x, &y);
```

Using the same scanf() statement as the example above, the input 5, 3.9, and no a, will result in the first value being read into x, then scanf() will stop executing as the expected character a is not found.

In some versions of c, the assignment suppressin operator(*) is supported. In the following example, if the input is Robert, 22 and 58289.97, then Robert will be read and assigned to name. The number 22 will be read but will not be assigned to age because of assignment suppression operator*, and 58289.97 will be read and assigned to salary.

```
char name[20]; int age, float salary,
scanf("%s %*d %f", &name, &age, &salary),
```

### 1.4 getchar()

The C library function getchar() can be used to input a single character into the computer. This function doesn't require arguments, but the parentheses are still a must. It returns a single character from a standard input device, usually the keyboad.

```
Syntax      int getchar(void);
```

In the following example, the first statement declares that c is a variable of type char. The second statement reads a single character from the standard inut dvice (usually the keyboard), and assigns the value to the variable c.

```
char t;
c = getchar();
```

### 1.5 putchar()

putchar() is a complementary function to getchar(). This is a standard output function provided by the stdio library. This transmits a single character to a standard output devie, usually the monitor. The character being output is usually of the char type and the variable name should be passed in as a parameter to the putchar() function. The parameters will be enclosed in the parentheses after the putchar().

```
Syntax      int putchar(int c);
```

In the following example, putchar() writes out a character to screen. The value of c is displayed as a character, c should be declared as a character variable or an integer variable. If it is declared as an integer variable, its value should be a valid ASCII value for a character.

The following is a simple program to read in a character from the keyboard and print it out in uppercase on the screen.

```
/* ..................................................................................... */
/*Name : char_conv.c-a program to read in a character and prints it out in uppercase */
/* Date : 5/29/97                                                                      */
/* Author : Prasad Putta                                                               */
```

```
/* ..................................................................................................... */
#include<stdio.h>
main()                              /*In C, only main function can exist without a type
                                                                    and not any other functions*/
{
        int  lower, upper;

        lower = getchar();
        upper = toupper(lower);
          /*toupper(name) is a function that converts lower case char to upper*/
        putchar(upper);
}
```

### 1.6 gets() and puts()

C contains a number of other library functions. For the input and output of character strings, the most appropriate and recommended functions are gets() and puts().

Each of these functions accepts a single argument. The argument must be a variable name of the type character string. A string may include whitespace characters and end with a newline character (i.e., when the Enter is pressed).

The following example prompts the user to enter their name. Let us say that the variable 'name' is a character string which can hold a maximum of 20 characters.

```
puts("Please enter your name");
```

The following example reads the input character string into the variable called name.

```
gets(name);
```

The following example displays the contents of the variable name to the screen.

```
puts(name);
```

## Chapter 4 - Exercises

1. What are commonly used input/output functions in C?
2. What is the standard header file included in C programs for input/output operations? How is it declared?
3. What is the difference between getchar() and gets() functions in C? Explain using two examples for each.
4. What is the difference between putchar() and puts() functions in C? Explain using two examples for each.
5. What is the difference between the printf() and scanf() functions in C? Explain using two examples for each.
6. What is a whitespace character?
7. What is a control string? Explain wih two examples.
8. What is the difference between the arguments that are passed to printf() and scanf() functions. Explain with examples.
9. What are conversion specifications? Explain with examples.
10. What is & and why is it used in scanf() functions? What happens if you do not use it? For what type of input data do you not use it?
11. What is precision with respect to printf() and scanf() functions? Explain with examples.
12. Summarize the meaning of the more commonly used conversion characters within the control string of a printf() function.
13. Summarize the meaning of the more commonly used conversion characters within the control string of a scanf() function.
14. What happens if an input data items contains either more of fewer characters than the maximum allowable field width? Explain with examples.
15. How can the assignment of an input data item to its corresponding argument be suppressed? Explain with an example.
16. What is the difference between the f-type, c-type, and g-type conversion characters in a control string of a printf() function. Explain with examples.
17. What happens if an output data items contains either more or fewer charaters than the maximum allowable field width? Explain with examples.
18. What happens to a floating point number if it must be shortened to confirm to a precision specification? What happens to a string? Explain with examples.
19. How can short integer, long integer, unsigned integer, and double precision arguments be indicated within the control string of a printf() function? Explain with examples.
20. Summarize the purpose of flags commonly used within the printf() function.
21. Can two or more flags appear consecutively within the same character group? Explain with examples.
22. Explain the difference between the printf() and putchar() functions, give examples.
23. Explain the difference between the scanf() and getchar() funtions, give examples.
24. Explain the use of precision in the scanf() statement, give examples.

## CHAPTER – 5

# 1. OPERATORS

In C there are several categories of operators. Some of the more important operators are : **arithmetic, unary, increment** and **decrement, relational, logical, assignment,** and **conditional.**

### 2.2. Arithmetic Oprators

Table 4.1 lists the arithmetic operators. Integer division truncates the fractional part. The modulus operator produces the remainder when you divide an integer.

| | |
|---|---|
| + | addition |
| – | subtraction |
| * | multiplication |
| / | division |
| % | modulus |

Table 4.1 Airthmetic Operators

The following examples give some arithmetic expressions and their computed values. For the examples, x and y are two decimal integers variables with values of 20 and 3.

| Expression | Value |
|---|---|
| x + y | 23 |
| x - y | 17 |
| x * y | 60 |
| x / y | 6 |
| x % y | 2 |

For the following examples, a and b are two floating point variables with values of 20.5 and 2.0.

| Expression | Value |
|---|---|

| | |
|---|---|
| a + b | 22.5 |
| a - b | 18.5 |
| a * b | 41.0 |
| a / b | 10.25 |
| a % b | illegal<br>modulus can only be performed on integers |

In C, operators are grouped hierarchically according to their precedence (i.e., the order of evaluation). Operations with a higher precedence are carried out before operations with a lower precedence. The natural order of precedence can be altered by using parentheses.

The arithmetic operators *, / and % are in one precedence group. The operators + and - are in another. The *, /, and % have a higher precedence than the + and - operators.

Associativity is another important consideration. Associativity refers to the order in which consecutive operations within the same precedence group are carried out. Within each of the precedence groups described above, the associativity is left-to-right.

If you have the float variables p = 1.0, q = 2, r = 3, and s = 4. What will be the value of the following expression?

```
p - q / r * s
```

According to the associativity rules, * and / are evaluated before -. Among * and / the associativity is from left - to - right, therefore the expressin is evaluated as p - ((q/r) * s) = -1.6666667.

**Parenthesis Rule :** Parenthesis alter the order of precedence. As stated earlier, operations within parentheses are carried out first. If you have a nested parentheses (one within the other), as in case of the above expression, then the innermost parentheses are evaluated first, then the next innermost, and so on. It is a good idea to use parentheses to clarify an expression. It is better to have extra parentheses rather than to have an incorrect expression. In complex expressions, care should be taken to use the correct number of parentheses in the correct locations.

Parenthesis have the highest order of precedence among all the operators and their associativity is from left-to-right.

Now suppose that you have a similar expression as in the previous example, but with parenthesis. This expression is (p-q) / (r*s) = -0.8333333, not -1.6666667. In this case the left parenthesis is evaluated, then the right, and then the quotient is determined.

```
(p - q) / (r * s)
```

### 1.2 Unary Operators

The operators that act on a single operand to produce a new value are called unary operators. Unary operators usually precede the single operand, however, sometimes the unary operator may be written after the operand. Some important unary operators supported by C are : unary minus, increment operator, decrement operator, size of, and cast.

#### 1.2.1 Unary Minus

The most common unary operator is unary minus. This is where a minus sign precedes a numerical constant, variable, or expression. In C, all numeric constants are positive thus, a negative number is actually an expression consisting of a unary minus and a positive numeric constant. It is important to note that the unary minus is different from the arithmetic minus which denotes a subtraction operator. The subtraction operator requires two operands. The following are a few examples of expressions using the unary minus operator.

-512

-0.25

-3E-7 -num1-(a+ b)

-5 * (p + q)

#### 1.2.2 Increment & Decrement Operators

The increment operator ++ adds one to the operand. The decrement operator -- subtracts one from the operand. These operators must be used either as prefix operators (before the variable, as : ++n) or as suffix operators (after the variable as : n++). In both cases the value of the operand (n) is incremented. The expression ++n increments n before its value is used, while n++increments n after its value is used. This difference is very significant when such operators are used in expressions.

For example, if n=10, then the statement x = n++ assigns a value of 10 to x and then increments n by one to 11. Therefore, x = n++; is the same as x = n; and n = n+1.

On the other hand, the statement x = ++n; first increments n to 11 and then assigns a valueof 11 to x. Therefore x = ++n; is same as x = n = n+1; or n = n+1; and x = n.

In an increment statement by itself, the position of the operand does not make a difference. The end effect of the statement is to increment the operand.

In the following example, the increment operator causes the variable count to beincremented by one.

```
count++;
  or
++count;
```

The decrement operator works in a similar fashion. For example :

```
#include<stdio.h>
void main()
{
        int i = 1;
        printf("i=%d\n",i);        /* i=1 will be printed */
        printf("i=%d\n", ++i);     /* i=2 will be printed */
        printf("i=%d\n", i);       /* i=2 will be printed */
        printf("i=%d\n", i++);     /* i=2 will be printed */
        printf("i=%d\n", i);       /* i=3 will be printed */
        printf("i=%d\n", i--);     /* i=3 will be printed */
        printf("i=%d\n", i);       /* i=2 will be printed */
        printf("i=%d\n", --i);     /* i=1 will be printed */
        printf("i=%d\n", i);       /* i=1 will be printed */
        return;
}
```

### 1.2.3 Sizeof

Sizeof is an operator used to compute the size of any object. The sizeof operator always precedes theoperand. The operand may be an expression or a cast the operation yields an integer equal to the size of of the specified type in bytes.

```
Syntax      sizeof(type);
```

In the following example, the operator returns two bytes, which is the sizeof integer.

```
sizeof(int);
```

The following operatio returns 10 * 2 = 20 bytes.

```
sizeof(10 * int);
```

Is the above example correct? Does your computer return two bytes for the sizeof integer? How can you find out the ize of each variable type? Try the following printf() statements in your own program on your computer.

```
printf("The size of integer on my computer is %d bytes.\n", sizeof(int);
```

```
printf("The size of float on my computer is %d bytes.\n", sizeof(float);
```

```
printf("The size of double on my computer is %d bytes.\n", sizeof(double);
```

```
printf("The size of char on my computer is %d bytes.\n", sizeof(char);
```

### 1.2.4 Cast

Thecast unary operator also precedes the operand. In general a reference to cast is written as type (variable). This is used to treat one type of variable as another desied type. For example, if x is a variable of type int, and for some reason you want to treat x as a float, you an cast it as follows :

```
float(x);
```

For example, suppose that x is a decimal integer whose value is5, and f is a float whose value is 2.2. The expression (x+f)%4 will be invalid, because (x+f) will be float =7.2 and 7.2% 4 will be illegal.

Note that in mixed-mode operations, as in the above example adding a integerwith a float, the lower datatype is converted into the higher type and then the expression is evaluated. In the above case, where you have (x+f), this will become 5.0+2.2, which will be 7.2.

On the other hand, if you cast the expression as int, then the expression will no longer be illegal and you will get the answer 3, It will be evaluated as int(x+f)%4, which is int(5+2.2) %4, which is int(5.0+2.2) %4, which is int(7.2)%4, which is 7% 4, which is equal to 3.

## 1.3 Relational Operators

Relational operators are used for comparing two quantities. The result of the comparison is true or false. Again, it is very important to note that in c, false is represented by an integer value zero(0) and true is represented by any non-zero integer value, usually 1. Table 4.2 lists the relational operators and their definitions.

| | |
|---|---|
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |

Table 4.2 Relational Operators

In the following example, both expr1 and expr 2 expressions are evaluated first and the results are compared. If the result of expr1 is less than the result of expr2, the expression returns true. If the result of expr2 is greater than the result of expr1, the expression returns false.

```
(expr1 < expr2)
```

all the relational operators fall in the same precedence group, which is lower than the unary and the arithmetic operators. Their associativity is from left-to-right.

### 1.4 Equality Operators

There are two equality operators, equal to and not equal to, which are associated with the relational operators. They are in a separate precedence group, beneath the relational operators and their associativity is from left-to-right. Table 4.3 lists the equality operators and their definitions.

| | |
|---|---|
| == | equal to |
| ! = | not equal to |

Table 4.3 Equality Operators

In the following example, the values int x =1, y = 2, z = 3, int j = 5, float f = 5.3 and char c = 'w' are used to evaluate the expressions.

| Expression | Interpretation | Value |
|---|---|---|
| x > y | false | 0 |
| (x+y)<=z | true | 1 |
| (y+z)>(x+7) | fale | 0 |
| x !=2 | true | 1 |
| y == 2 | true | 1 |
| f > 5 | true | 1 |
| (f+j)<=10 | false | 0 |
| c == 119 | true | 1 |
| c! = 'x' | true | 1 |
| c>=20 * (j+f) | false | 0 |

Note that the above example uses the ASCII charaacter set to get the value of the variable chara c.

### 1.5 Logical Operators

C provides two logical operators, also called logical connectives : **logical AND (&&) an dlogical OR(||).** these are used to connect expressions. Each of the logical operators is in its own precedence group. Logical AND has a higher precedence than logical OR. Expressions connected by the logical operators are evaluated left-to-right. Table 4.4 lists the logial operators and their definitions.

| | |
|---|---|
| && | logical AND |
| \|\| | logical OR |
| ! | logical NOT |

Table 4.4 Logical Operators

The logical operators act upon operands that are themselves logical expressions. In other words, the individual logical expressions are combined with either logical OR, or logical AND operators to form one complex expression which will result in either true or false. The result of a logical AND is true only when both the operands (the logical expressions connected by the logical AND) are true. On the other hand, the results of a logical OR are true when either/or both of the operands are true.

For example, in the statement below, expression 1 is first evaluated and if its result is true, then expression 2 is evaluated. If expression 2 is true, then the entire statement is true, otherwise it is false. If expression 1 is false then the entire expression is false and expression 2 is not evaluated.

```
if (expressional && expression2)
```

In the following example, expression 1 is first evaluated and if its result is true, then the entire expressionis true and expression 2 is not evaluated. If expression 1 is false then the entire expression is false and expression 2 is not evaluated.

```
if (expressional && expression2)
```

Inthe following example, expression 1 is fist evaluated and if its result is true, then the entire expression is true and expression 2 is not evaluated. If expression 1 is false then expression2 is evaluated and if it is true, then the entire expression is true, otherwise the entire expression is false.

```
if (expressin 1 ||expression2)
```

C also includes the unary operator referred to as logical negation or logical NOT. This operator negates the value of a logical expression, that is, if an expression is true it becomes false and vice versa. In Table 4.5 !x stands for not x, which means that when x is true, not x is false and vice versa.

| logical expression x | logical expression y | x && y | x \|\| y | !x | !y |
|---|---|---|---|---|---|
| true | true | true | true | false | false |
| true | false | false | true | false | true |
| false | true | false | true | true | false |
| false | false | false | false | true | true |

Table 4.5 Truth Table

To evaluate the following examples, use the values int j = 5, float f = 5.3 and char c = 'W'. The arithmetic operations will be carried out before the relational or euquality operations, wich are carried out before logical operations.

In the last expression, the addition operation (i.e., j + f) is carried out first, then the relational comparison (i.e., c! = 'C'), and finally, the logical OR condition.

| Expression | Interpretation | Value |
|---|---|---|
| f>5 | true | 1 |
| l(f>5) | fale | 0 |
| J<2 | false | 0 |
| !(j<2) | true | 1 |
| !(c=='w') | true | 1 |
| f> 2.3 &&j<7 | true | 1<br>parentheses not necessary<br>natural order of precedence |
| (j<2)&&(c=='w') | false | 0<br>parentheses not necessary,<br>but okay |
| j<2\|\|c == 'w' | true | 1 |
| c! = 'C' \|\| j+f<2 | true | 1 |

**Short-Circuit Evaluation :** The logical expressions joined by logical AND and logical OR are evaluated left-to-right, but only until the overall true/ false value has been established. Thus, a complex expression will not be evaluated in its entirety if its value can be established from its constituent

operands. Or in other word, it may not be necessary to evaluate the right operand at all in expressions with the logical AND. If the left operand of the logical AND is a false, the whole expression returns false.

Similarly, if the left operand of the logical OR is true, the whole expression returns true. In this case the right operand is not evaluated. This is called short-circuiting the boolean expression. This is something to keep in mind when programming with logical expressions.

Sometimes, C programmers may rely on a short-circuit evaluation to prevent an evaluation that would cause an error.

In the following example, we want to evaluate an expression containing the sqrt() function, which is not defined for negative numbers. Because of short-circuit evaluation, the right operand is only evaluated when x is a non-negative number, thus preventing an error.

```
(x>=0.0) && (sqrt(x) <25.0)
```

### 1.6 Assignment Operators

C provides several assignment operators. All of them are used to form assignment expressions or assignment statements which assign the value of an expression to an identifier or variable. You have seen various assignment operators used in the assignment statements section earlier. The most commonly used assignment operator is=.

```
Syntax      identifier = expression
```

Some examples using the = assignment operator are :

```
x = 3
a = b
sum = x + y
area = PI * radius * radius
```

Note that the assignment operator = should not be mistaken for the equality operator ==. The assignment operator is used to assign a value to an identifier, whereas the equality operator is used to determine if two expressions have the same value.

If you assign a particular value to a variable of a different type, the value is converted into the type of the variable and assigned.

For example, if x is a decimal integer and you assign a value of 2.2 to x, then 2.2 is truncated to 2 and this value is assigned to x.

Similarly if f is a float and you assign a value of 2 to f, the value is converted to 2.0 and assigned to f.

Under some circumastances, this automatic type conversion may result in an alteration of the data. Careless type conversions are sources of errors. Some possible alterations are as follows :

* A floating-point value may be truncated if assigned to a decimal integer variable or identifier.
* A double precision value may be rounded if assigned to a single-precision (float) identifier.
* An integer quantity may be altered if assigned to a short, or to a char type identifier (some high-order bits may be lost).
* The value of a charater constant assigned to a numeric-type identifier will depend on the particular character set in use, and may cause inconsistencies from one version of C to another.

Table 4.6 lists some other assignment operators provided by C.

| Operator | Definition |
|---|---|
| += | sum equal to |
| -= | substraction equal to |
| *= | multiplication equal to |
| /= | division equal to |
| %= | percentage equal to |

Table 4.6 Assignment Operators

The general format of these types of expressions are :

| Syntax | expression1 assignment operator expression2 |
|---|---|

In the following example, int j = 3, k = 6, float f = 2.5, and g = -3.25.

| Expression | Expression Equivalent | Final Value |
|---|---|---|
| j f =3 | j = j +3 | 6 |
| g -= g | g - g - f | -5.75 |
| j *=(k+2) | j=j*(k+2) | 24 |
| f/=5 | f - f/5 | 0.5 |
| k% = (j+3) | k =k%(j+3) | 0 |

## 1.7 Conditional Operator

C provides a conditional operator? A conditional expression is an expression that makes use of the conditional operator.

| Syntax expressional ? expression2 : expression3 |
|---|

With this operator, expression1 is evaluated first. If expression 1 is true, then expression2 is evaluated and the result becomes the value of the conditional expression otherwise, if expression 1 is false, expression 3 is evaluated and the result becomes the value of the conditional expression.

In the following example, (a>b) is evaluated first. If a is greater than b then you get true, otherwise false. If true, the right hand expression gets the value of expression a, otherwise the right hand expression gets the value of expression b, which in turn is assigned to max. In other words, if a is greater than b, max will be equal to a, otherwise max will be equal to b.

| max = (a>b)? a:b |
|---|

The conditional operator has its own precedence, just above the assignment operators with right-to-left associativity.

## 1.8 Precedence and Associativity of Operators

Table 4.7 lists the associativity of operators. It also includes some operators that will be dealt in future chapters. Unary +, - and * have higher precedence than the binary forms. Operators in the same row of the table haave the same precedence; however rows are in order of decreasing precedence, i.e. */and % have precedence which is higher than that of +, or <<, >>.

| **Operators** | **Associativity** |
|---|---|
| ( ) [ ] . -> ++(postfix) -(postfix) | left to right |
| ++ (prefix) --(prefix)! ~ size of (type) + (unary) - | right to left |
| (unary) & (address) * dereference) | |
| * / % | left to right |
| + - | left to right |
| << >> | left to right |
| < < > >= | left to right |
| += != | left to right |
| & | left to right |
| ^ | left to right |
| \| | left to right |
| && | left to right |

| | |
|---|---|
| \|\| | left to right |
| ?. | left to right |
| + - * /= % - >>= << & = ^ - \| = | right to left |
| , (comma operator) | left to right |

It should be noted that operators of the same precedence generally are evaluated from left-to-right with three exceptions. The operators(), [] and -> have the highest precedence level.

In the following example, we have / and * of the same precedence which have higher precedence than +. Since the / operator is first, the compiler evaluates c / d first and the result is multiplied by e. The resulting value is assigned to a.

```
a = b + c / d * e;
```

In the following expression, b is incremented first, then the result is multiplied by e and the result is added to c, and this value is assigned to a.

```
a = ++b * e = c;
```

## CHAPTER – 6

# 1. CONTROL STATEMENTS

The order, or sequence of execution of statements in a program is referred to as control flow or flow control. In any programming language there are three types of control flow : sequential, **conditional,** and **iterative** or looping.

In the following sections, we will look at some of the control statements C provides to modify control flow. It is critical that you as a programmer, whether a C or database programmer, clearly understand control flow and control statements.

## 1.1 Sequential Control

As shown in Figure 5.1, control flows from one statement to the next in sequene in seqnential control flow. By default, control flows in a seqential order unless a control statement changes the flow.

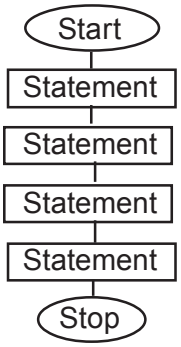



**Figure 5.1 Sequential Control Flow**

To illustrate sequential control flow, let's use the radius/area of a circle (circle.c) program from Chapter 3 (p.27). Normally the program executes each statement once, in order, and then exits. In other words, the program executes the following steps :

1. prompt user for radius
2. read input value
3. compute area of circle
4. display results
5. exit

## 1.2 Conditional Control

You may not always want to execute a program statement-by-statement. Sometimes, you may want to perform a specific task if a condition is true, or perform a different task if a conditionis false. To do this you use a condition based statement to control the flow.

A conditional statement is also called a selection statement, which is used for selectivity executing statements from among one or more alternatives. The selection is based upon the value of the control expression or condition. C provides three basic types of selection or conditional statements : if, if-else, and the switch statement. The conditional operator which you studied in the previous chapter, works like an if-else statement.

### 1.2.1 If Statement

I statements ae used to execute a statement, or block of statements provided the control expression or the condition returns true.

| | |
|---|---|
| Syntax | if (expression) statement; |
| | if (expression)<br>{<br>statement;<br>statement;<br>statement;<br>} |

As shown in Figure 5.2, the condition or control expression is evaluated first. If the value of this expression is true, then the controlled statement is executed. If the value is false, the statement is ignored.

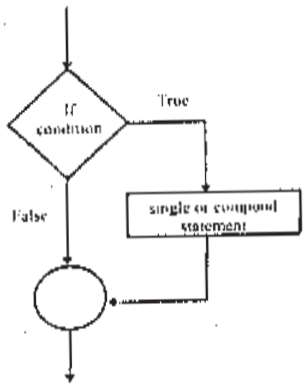



**Figure 5.2 If Statement Control Flow**

Using the circle. c example again, let's say that you want the program to compute the area only if the input radius is an integer, otherwise you want it to exit. The program should execute the following steps :

1. prompt user for radius
2. read input value as radius
3. check whether the input radius is an integer

   If the radius value is an integer (true), then the program will :

4. compute the area of the circle
5. display the results
6. exit

   If the radius value is not an integer (false), then the program will :

4. exit

### 1.2.2 If-Else Statement

If-else statements are used to provide two different options for statement execution based upon the comparison of an input value to a condition.

| | |
|---|---|
| Syntax | if (expression)<br>statement1;<br>else<br>statement2; |
| | if (expression)<br>{<br>block1;<br>}<br>else<br>{<br>block2;<br>} |

As shown in Figure 5.3 the condition or the control expression is evaluated first. If the expression is true, then the code within the first block or statement 1 is executed, on the other hand, if the expression is false then the second block or statement2 following the else is executed.

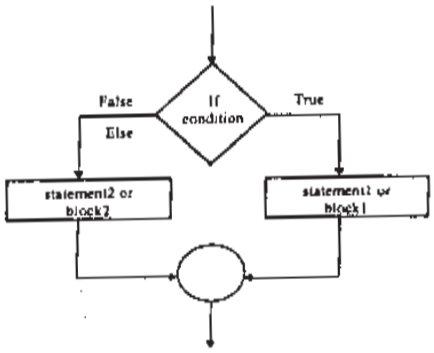



**Figure 5.3 If-Else Statement Control Flow**

Using the circle.c example again, let's say that you want the program to compute the area only if the input radius is an integer, otherwise you want it to display an error message and exit. The program should execute the following steps :

1. prompt user for radius
2. read input value
3. check whether the input radius is an integer.

   If the radius value is an integer 9true), then the program will :

4. compute the area of the circle
5. display the results
6. exit

   If the radius value is not an integer 9false), then the program will :

4. display an error message
5. exit.

Some points to be aware of when using the if-else sturucture :

- There is no semicolon after the expression.
- There can only be one statement after the if part, and one statement after the else part. If you have more than one statement, enclose them in braces and make one compound statement.
- the else part is optional, but if you have the else, it needs to be followed by a statement.

As shown in the following example, a semicolon by itself is considered a null statement. Here the area is computed irrespectiveof the expression (radius!=0). If (radius!=0), then a null statement is executed.

```
if (radius!=0);
        area = PI * radius * radius;
```

### 1.2.3 Conditional Expression

The conditional expression formed using he conditional operator(?) is very similar to the if-else statement. Using this structure, expression 1 is evaluated and if it is true (or non-zero), then expression2 is evaluated and that is the value of the conditional expression. Otherwise, expression 3 is evaluated and that is the value of conditional expression. Only one of the expressions, either expression 2 or expression3, is evaluated.

```
Syntax    expression ? expression 2 expression 3
```

A conditional expression can be used to find the larger number among two numbers as shown in the following example.

```
max   (a > b) ? a:b;
```

Proper use of a conditional expression can improve readability and program simplicity. For instance, if the above example was written as an if-else statement, it would look like the following :

```
if ( a>b )
        max = a;
else
        max = b;
```

### 1.2.4 Nested If Statement

As you have seenin the previous section, the if statement, including any block statements inside, is considered to be one statement. This is true for the if-else statement too.

In addition, if and if-else statements can themselves be the statement for another if or-if-else statement. These are called nested-if statements.

```
If (expr1)
      if (expr2)
         if (expr3)
            {
               .............
               .............
               .............
            }
      else
            {
               ............
               ............
            }
      else
            {
               ............
               ............
            }
else
      {
      .........
      }
```

The else part of an if-else is optional, but there is an ambiguity when an else is omitted from a nested if sequence. This is resolved by associating the else with the closest if lacking an else.

In the following example, there is a question as to which if the else belongs to, the first or second if. This problem is called the dangling else problem. In this case, the else goes with the inner if.

```
if ( n>0 )
      if ( a > b )
            z = a;
      else
            z = b;
```

If you want the else to go with a different if, braces can be used to force the proper association. Using the same example, braces group the else with the outer if as shown below.

```
if ( N > 0 ) {
        if ( a > b )
        z = a;
else
        z = b;
```

Using braces is a good practice. It is better to have an extra set of braces rather than risk associating the else with an incorrect if statement and ending up with the dangling else problem.

### 1.2.5 Else-if Statement

It is also possible to have an if statement as the inner part of an else. This is called an else-if statement and it allows you to select from multiple program paths based on a condition.

Syntax

```
if ( expression1 )
        {
                statements;
        }
else if ( expression2 )
        {
                statements;
        }
else
        {
                statements;
        }
```

The else condition is optional. It is more like the default case and is executed if both expression 1 and expression 2 are false. If expression 1 is true, then the code within the block following the if is executed and remaining conditions are skipped. If expression 1 is false then the test is performed to see if expression2 is true and if so, the next block of code is executed. There is no limit on the number of else-if conditions that can be used.

For example :

```
if      ( not raining )         I will go home
else if ( Any one has an umbrella ) I will go to Pizza Hut
else    I will have coffee here
```

For example :

```
if (n < 10)
        printf("You entered a single digit number\n");
else  if  (n<100)
        printf("You entereda double digit number\n");
else  if  (n<1000)
        printf("You entered a triple digit number\n");
else
        printf("You entered a number with four or more digits\n");
```

Indentation is for the readability of the user, not for the computer. It doesn't make any difference in the execution of the program. The above example can be written in a different way using different indentation as shown in the following example. The program will execute in exactly the same way.

```
if  (n<10)
        printf("You entered a single digit number\n");
else if (n<100)
                printf("You entered a double digit number\n");
        else if (n<1000)
                printf("You entered a triple digit number\n");
                else
                printf("You entered a number with four or more digits\n");
```

### 1.2.6 Switch Statement

Another approach C provides for multiple selections is the switch statement.

```
switch (expression)
{
case constant_expression : statements;
case constant_expression : statements;
default                  : statements;
}
```

The expression following the keyword should be an integer valued constant or a constant expression. All case expressions should be different. If a case matches the expression value, then the execution starts at that case and continues to the bottom, executing all statements.

If only those statements within a case are to be executed, and the rest of the statements in other cases are to be skipped, then a break statement should be used.

```
switch (expression)
{
case constant_expression : statements;
                           break;
case constant_expression : statements;
                           break;
default                  : statements;
}
```

The default is optional. If it is present, then statements within the default case will be executed when all the cases mentioned are not satisfied.

For example :

```
switch (value)
{
case 1:   addition = a + b;
          printf("Sun=%d", addition);
          break;
case 2:   subtraction  = a - b;
          printf("Difference = %d", subtraction);
          break;
case 3:   multiplication = a * b;
          printf("Product = %d", multiplication);
          break;
case 4:   division = a / b;
          printf("Quotient = %d", division);
          break;
case 5:   remainder = a%b;
          printf("Remainder = %d", remainder);
          break;
default :
          printf("Please enter a valid value\n");
}
```

### 1.3 Iteration Statements

You may want to repeat certain instructions again until a specific condition is satisfied. This requires an iteration or looping structure. The group of instructions that are repeated is called a block. There are three looping structures in C: the while loop, the for loop and the do-while loop.

### 1.3.1 While Loop

A while loop is usually used if the number of times a certain block of code is to be repeated is unknown at the time of coding. The condition for ending a while loop is usually generated through the execution of code within the block to be repeated.

```
Syntax    while (expression)
                statement;
          while (expression)
                {
                        code to be repeated
                }
```

The expression in the while loop is called the conditional expression or control expression. The vaiable in the conditional expression is called the control variable.

As shown in Figure 5.4, the user is prompted for input, the input is read in and evaluated against the condition or control expression. If the condition is true, then the code within the block is executed. The user is again prompted for input and control returns to the beginning of the block where the condition is evaluated again.

The block of code will be repeated for so long as the condition returns true. When the condition is false, the program will exit.

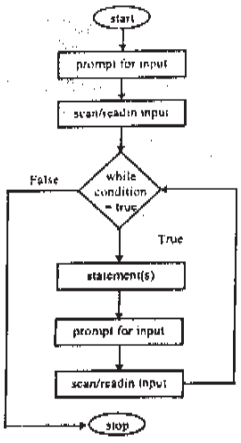



**Figure 5.4 While Loop**

Using the circle example again, let's say that you want the progam to continue computing the area for so long as the input radius is not zero. The program should executed the following steps:

1. prompt user for radius
2. read input value
3. check whether the input radius is a non-zero.
   If the radius value is not zero (true), then the program will;
4. compute the area of the circle
5. display the results
6. prompt user for next value
7. read input value
8. return to beginning of while loop and test the condition again
   If the radius value is zero (false), then the program will
4. exit

Some points to be aware of when using while loops:

- Because the conditional expression is evaluated before the program enters the while loop, the control expression should be initialized before use to avoid run time errors.
- It is very important to update the value of the control variable in each iteration of the loop. Otherwise, you may not reach the exit condition and you may end up in an infinite loop.
- The maximum number of times that a while loop executes depends on the control expression and the control variable. The minimum number of times that any while loop can execute is zero i.e., when the control expression returns false the first time.

### 1.3.2 For Loop

When the number of times a block of code is to be repeated is known at the time of coding, then a for loop can be used. In the for loop, the initializing statement, exit condition statement, and the increment statement are all specified together at the top.

Therefore, the statement or the block of code to be repeated will not contain the increment statement.

```
Syntax for (initializing statement; exit condition statement, increment statement)
        {
           code to be repeated
        }
```

The following steps explain how the for loop works :

1. The initialization statement is evaluated.
2. The exit condition statement is checked, if it is true (or non-zero) then the block of statements of the for loop is executed. Then step 3 is executed. If the exit condition statement fails (results in false), then the loop is not entered, control passes to the statement immediately after the for loop. Step 3 is not executed.
3. After the statements of the for loop block are exeucted, the increment statement is execute (the control variable is updated). Then, steps 1 and 2 are repeated until the exit condition results in false.

The following example illustrates the steps used inthe for loop.

1. i = 0; is evaluated and the value 0 is assigned to the variable i.

2. i<5; (the exit condition statement )is evaluated. Since the value of i is 0, which is less than 5, the exit condition statement results in true. Therefore, control passesthrough the loop, which means that the for loop's statements will be exeuted.

3. i++; is executed (i is incremented by 1) resulting in the value of i = 1. Then, i < 5; (the exit condition) is evaluated. Since the i value is 1, which is less than 5, the exit condition results in true. Therefore, control passes through the loop again. This process continues until the value of i becomes 5, at which time the exit condition statement will fail. Hence, the loop is executed five times.

```
for (i=0; i<5; i++)
        {
            printf("THE VALUE OF i IS %d \n", i);
        }
```

Note that in the above example, if the exit condition statement is changed to i <=5, then the loop is executed 6 times and not 5. Because of this, care should be taken in forming the initializing statement and the exit condition statement. Also, the increment statement may increment the value of the control variable by one or more. For example, you can have i = i + 2; as the increment statement. Also, you can have a decrement statement instead of an increment statement. For example the above for statement can also be written as:

```
for (i=5; i>0; i--)
        {
        ptintf("THE VALUE OF i IS %d \n", i);
        }
```

The above example may also be written using a while loop. The choice of the type of the loop depends upon the programmer's preference.

### 1.3.3 Do-While Loop

A do-while loop is similar to the while loop. If you are sure tha the block of statements of the loop is to be executed at least once, then the do-while loop can be used. Code within the do-while loop is executed until the value of the expression becomes false or zero.

In a do-while loop, the code is first executed and then the decision to continue or exit the loop is made. Here the condition is checked at the bottom and not at the top as in the while loop. Because of this, the do-while loop is also called a bottom checking loop and the while loop is called a top checking loop.

Control passes through the do-while loop and each statement of the loop is executed. Then the conditional expression is checked at the bottom, and if it results in true (or non-zero), then control goes back into the loop and iteration continues. The loop is executed until the condition fails.

Because the conditional expresion is checked at the end of the block, the do-while loop is always executed at least once.

```
Syntax      do
            {
            code to be repeated
            }  while (expression);
```

For example :

```
i = 0 ;
n = 5;
do
      {
      printf("THE VALUE OF i IS %d \n", i);
      i++;
      } while (i<n);
```

## 1.4 Jump Statements

Sometimes you may not want to continue the loop until the end. Or, if a certain situation or condition occurs, you may want to skip that particular iteration. C provides structures for skipping over an instruction(s) in a loop. This is done through the use of jump statements. Table 5.1 summarizes

the types of jump statements covered in this section and their purpose.

| | |
|---|---|
| break | emergency exit out of the loop--overrides the exit condition check of the loop |
| continue | skips the execution of the remaining statements in that iteration and continues with the next iteration (based upon the exit condition) |
| goto | unconditionally the control jumps to the statement after the label of the goto statement |

Table 5.1 Jump Statements

### 1.4.1 Break Statement

The break statement is used to exit the loop. It is a sort of emergency exit. When break statements are used in loops, the loop is exited and the program flow is transferred to the next statement after the loop. It can be used in iterative loops and also in switch statements to exit a case as shown earlier.

In the following example, if the expression of the if statement is true, then the break statement is executed and the for loop is exited without executing the remaining statements in the for loop. The syntax of break is similar for while and do-while loops.

```
for (i=0; i<10; i++)
        {
                stateament(s);
                if (expression)
                    break;
                statment(s);
        }
```

### 1.4.2 Continue Statement

The continue statement is only applicable to loops. When a continue statement is executed, the next iteration of the enclosing loop is started, skipping the remaining statements in the loop.

In the following example, if he expresion is true, then the continue statement is executed and it causes the next iteration of the enclosing for loop to begin, i.e. the increment step is executed. In the while and do-while loops, this would cause the test part to be executed.

```
for (i=0; i<10; i++)
        {
                if (expression)
                  continue;
                statements;
        }
```

### 1.4.3 Goto Statement

The worst jump statement is the goto statement. A goto statement uses an identifier or label, and the program can be made to jump to that label. A label can be declaread anywhere in the function and is followed by a colon.

```
void function_a( )
{
        declarations;
        statement 1;
        statement 2;
        AGAIN :
        statement 3;
        goto AGAIN;
        statement 4;
}
```

### 1.4.4 Return Statement

Return statements provide a way of exiting the functions. Return statements can be used anywhere in functions as shown below. If the return statement is executed, then the function is exited skipping the remaining statements if any.

```
Syntax    void function_a( )
              {
                declarations;
                statements;
                if (expression)
                        return;
                statements;
                return;
              }
```

## Chapter 6 - Exercises

1. Repetitive action is essential in many situations. Therefore, a programmer must know precisely how a while loop works. Study the following code in detail, then write down what you think gets printed. Then write a test program to check your answer.

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

## CHAPTER – 7

# 1. FUNCTIONS AND STORAGE CLASSES

A function is a self-contained program segment that carries out some specific, well-defined task. It is used to define a procedure or sequence of steps to be executed. In this chapter we will discuss the advantages of using functions and the three important function components: the function prototype (declaration), function call and function definition.

The storage class of a variable gives you its permanence and scope within the program. In this chapter we will cover the four storage classes : automatic, static, external, and register.

### 1.1 Advantages

Functions make a program more modular. Any big task or problem can be broken down into several subtasks, each of which can be represented by a function. Dividing a big program into several functions makes the program easier to read, understand, manage, and maintain. Also, irrelevant details can be buried in the functions and the chance of unwanted interactions can be minimized.

Functions enable programmers to build on what others havedone instead of starting over from beginning. Functions from one program can be used in other programs. Also, sometimes you need to repeat code many times at different locations in the program. Intead of writing the same code over and over, the programmer can write the code once as a function, then call the function when and where needed.

Functions can be used to hide operating details from the parts of the program that don't need to know about them. With well designed functions, users do not need to know about the functions implementation. They just need to know the function's purpose.

### 1.2 Function Prototype

The function prototype defines the function's name, the parameters that are passed into the function and the type of value the function returns. Prototypes should be placed at the top of a file, above all functions, or in include files.

| Syntax | return_type function_name (type_1, type_2, type_3,.... type_n); |
|---|---|

Functinos, like variables, are given names. Let's say you are writing a function to add together two numbers. You might give it a good descriptive name such as "add_two_numbers."

Information is passed into the function via special identifiers called arguments or parameters. The function input variables (parameters) may instead be defined inside the function. For example, first_number and second_number would be defined inside add_two_numbers. It is possible, however, to havae a function that has no parameters. Also you should note that variable names are not required, The C compiler will process the function so long as the variable types are given.

A function can be of any data type. A function is given a type based upon the type of value it returns. For example, if first_number and second_number are of the type int, then add_two_numbers will be of the type int. A function which doesn't return a value but just executes certain steps will be of the type void. It is essential that the function variable type match the program variable type that the function returns data to.

For example :

```
int add_two_numbers(int, int);
```

This can also be written as :

```
int add_two_numbers(int first_number, int second_number);
```

The function will pass information out in one of two ways: via the return statement, with a maximum of one value returned; or, via a scanf() or printf() function or multiple values. However, the function does not have to return a value, it can just perform a series of steps.

### 1.3 Function Call

A function call happens when the program accesses (or calls) the function. A function can be called from anywhere within the main() functin, or from within another function (depending upon where the prototype is, and where it is defined).

If the function returns a value there should be a corresponding variable of the same type to receive it. For example, the function add_two_numbers can return a value to the program variable total of the type int. The actual parameters are passed in as part of the function call.

For example, say you have two decimal integers num1, and num2 and you want their sum which will be stored in another decimal integer total. The functioncall will look like :

```
total = add_two_numbers (num1, num2);
```

As stated in section 6.2, a functin of the void type doesn't return a value. The call statement for a void function will include only the functin name with the actual parameters, if any.

In the following example you have a function with a prototype that prints the menu.

```
void print_menu();
```

This doesn't take in any parameters but you still have to have the open and close parenthesis. Since it returns no data it is of the type void. the function call for this function will look like :

```
print_menu();
```

There are two types of function calls : call by value and call by reference.

### 1.3.1 Call By Value

In call by value, formal parameters rather than actual parameters are passed into the function. This means that the called function is given a temporary variable (copy of the values) rather than the original. The original values remain unchanged regardless of how the function changes the copies.

In the following example, the parameters base and n are used as temporary variables. Whatever is done to n inside the function power has no effect on the variable that was passed to the function.

```
int power (int base, int n)
        {
        int p;
        for (p=1; n>0; --n)
        p - p*base;
        return p;
        }
```

The following is a program to add two decimal integer numbers. This program will prompt the user to enter two numbers, compute their sum and display it. A function is defined for computing the sum and is called in the main(). This example is used to show the different components of a function and call by value.

```
/* ..................................................................... */
/* Name : sum.c - This program will prompt for two decimal integers and displays*/
/*              their sum                                               */
/* Date : 5/29/97                                                       */
/* Author : Prasad Putt                                                 */
/* ..................................................................... */
#include<stdio.h>          /*standard input/output file used for printf( ) and scanf()*/
/* global variables declaratin*/                          /*no global variables*/
int add_two_numbers(int number_one, int number_two);   /*function prototype*/
void main()                                                  /*main functin*/
                 /* main function is void type, meaning that it will not return any data
                              item but will only execute the statements in the block*/
{
/*local variable declaration for the main( ) function*/
    int  num1, num2;                                        /*input variables*/
    int total;                                              /*output variable*/
/* get input data from the user*/
   printf("Enter any two decimal integer values for which you want the sum to be
           computed/n");
   scanf("%d %d", & num1, & num2);
   total = add_two_numbers(num1, num2);                      /*function call*/
                                                            /*print the sum*/
   printf("The sum of %d and %d is %d\n", num1, num2, total);
   return                                  /*returns control to the operating system*/
}
int  add two numbers (int number one, int umber two)       /*function definition*/
{
/*local variable declaration for the add_two_numbers( ) function*/
    int  sum;                                                /*output variable*/
    sum =number_one+number_two;                             /*compute the sum*/
   return sum;                                                /*return sum*/
}
```

The following is an example program demonstrating external (global) variable declaration and prototype.

```
/* ..................................................................... */
/* Name : glo_func.c - This program demonstrates the concept of global vs.   */
/*            local variables                                           */
/* Date : 5/2/97                                                        */
/* Author : Prasad Putta                                                */
/* ..................................................................... */
#include<stdio.h>      /*standard input/output file used for printf( ) and scanf( )*/
/*global variables declaration*/
int a = 1, b =2, c = 3;
/*function prototype*/
int  func(void);  /*void within the parenthesis indicates no parameters passed */
```

```
 /*main function */
void main( )
{
        printf("The value of func=%d/n The value of a, b, c after function call:\n",
                func( ) );
                                                        /* The value of func = 12 */
        printf("a=%d, b=%d,  c=%d\n",  a,b,c);
/* the values of a, b, c after function call are printed */
                        /* the values will be : a = 4, b = 2, c = 3 will be printed */
        return;
}
/*functin definition*/
int func(void)
{
        int b, c;
                                                    /* b and c are local variables*/
        a = b = c = 4;                      /*changes global a and local b and c*/
        return (a+b+c);
}
```

### 1.3.2 Call By Reference

In call by reference, actual parameters are passed into the function, meaning hat the called functin is given the original values rather than copies of the values. if the user makes any changes to the values in the function, these changes are reflected in the calling function also.

For the following example, the prototype for the function swap( ) is;

```
void swap (int *, int *);
```

In the following example, the addresses of the variables are passed to the function. Whatever changes occur to the values of the variables in the swap function, these changes are reflected in the main program.

```
main( )
        {
        int a, b;
        .....
        .....
        swap (& a, & b);
        .....
        }
void swap (int *c, int * d)
{
.....
.....
}
```

The following is a program to add two decimal integer numbers. This program is a modification of the above program which will demonstrate a function call by reference. This program uses a function called get_input() which will prompt the user to enter two numbers. This function is of the void type, but the variables are declared as global variables and hence the function call is by reference. Also, their sum is computed using the function compute_sum() which is called by value and will return the sum to the main( ) function. There is a third function of the void type to display the results. This is called display_results( ) and takes in one parameter, the total.

```
/* ..................................................................................... */
/* Name : sum.c - This program will prompt for two decimal integers and displays*/
          their sum
/* Date : 5/29/97
/* Author : Prasad Putta                                                              */
/* ..................................................................................... */

#include<stdio.h>          /*standard inut/output fil used for printf() and scanf() */
/*do global variables*/
int num1, num2;

/* function prototype */
void get_input( );                                          /* for getting the input*/
int compute_sum(int number_one, int number_two);
void display_results(int result);

/*main function*/
void main()
{
/*local variable declaion for the main() function*/
   int  total;                                                  /*output variable*/
/*function calls*/
get_input();
total = compute_sum(num1, num2);
displa_results(total);

return; /*returns the control to the operating system*/
}

void get_input()                                            /*function definitions*/
{
/*get input data from the user*/
printf("Enter any two decimal integer values for which you want the sum to be
computed\n");
        sanf("%d %d", & num1, & num2);
```

```
        return;
}

int compute_sum (int number_one, int number_two)
{
/*local variable declaration for the compute_sum() function*/
        int sum;                                                    /*output variable*/

        sum = number_one + number_two                       /*compute the sum*/

        return sum;                                                 /*return sum*/
}

void display_results(int result);
{       /* here num1 and num2 are global variables and only the result is passed*/
        printf("The sum of %d and %d is %d\n", num1, num2, result);
        return;
}
```

## 1.4 Function Definition

The function definition, as the name implies, defines the procedure, or the series of steps to be executed when the function is called.

```
Syntax    return_type function _name(argument declarations)
          {
          function code
          }
```

**For example :**

```
int add_two_numbers (int first_number, int second_number)
{
      int  sum;                    /*declaration of local variables for this function*/
      sum = first_number + second_number;/*steps to be executed in function*/
      return sum;                        /*value to be returned to calling function*/
}
```

The programmer can write their own functions. The return type and arguments to the function is optional. Values can be input to functions through the argument list or through global variables. Output from functions can be obtained through the return value of the function, through the arguments, through the global variables, or through any combination of these three.

You can also use library functions written by you or someone else. To do this you use the #include statement, followed by the header file name

which contains informatin about the location and the name of the library file. for example, the most commonly used functions provided by C are printf() and scanf(), the definitions for which are gotten through #include<stdio.h>.

### 1.5 Return Statement

Return statements provide a way of exiting functions. The return statement is the mechanism for returning a value from the function. Any expression can follow the return, but should have the same type as the function's return type. the expression can also be a constant.

```
Syntax      return variable;
            return expression;
```

For example :

```
return sum;
```

or

```
return (first_number + second_number;);
```

The expresion's value is returned and the function is exited after the execution of the return statement. An empty return statement without any expression is legal if the function's return type is void.

For example :

```
void function_a()
{
        declarations;
        statements;
        if (expression)
        return;
        statements;
        return;
}
```

Return statements can be used anywhere in functions. If the return statement is executed, then the function is exited skipping the remaining statements.

### 1.6 Storage Classes

Variables ae characterized in two ways, by datatype and storage class. As we have already discussed, datatype refers to the type of information represented by a variable. Storage class refers to the permanence of a

variable and its scope within the program. Scope is the portion of the program in which the variable is recognized. C has four different storage classes: automatic, static, external and register. Table 6.1 lists these storage classes and their scope.

| Storage Class | Keyword | Scope | Duration |
|---|---|---|---|
| automatic | auto | local | temporary |
| register | register | local | tempoary |
| static | static | local | persistent |
| external | extern | global (all files) | persistent |
| external static | static | global (one file) | persistent |

Table 6.1 Storage Classes

### 1.6.1 Automatic

Automatic variables are those variables defined within a functin or a block. The scope of these variables is limited to the function/block in which they are defined. Automatic variables come into existence only when the function/ block is executed and disappear after the execution of the function/block. Automatic is the default storage class for variables and hence he keyword auto is not specifically mentioned.

### 1.6.2 Register

The register storage class specifies that a variable of the automatic storage class be stored in a processor register (insted of main memory) if possible. the variables stored in registers are accessed much faster than those stored in the main memory. the appropriate use of register storage to speed up execution is highly implementation dependent. In appropriate use of register storage may slow down the executing code by interfering with the compiler's attempts to optimize registerusage.

| Syntax | register variable_type variable_name9(s); |
|---|---|

For example :

```
register int i, j, k;
```

### 1.6.3 Static

A variable can be made to retain its value after a block's execution is completed by giving a static qualifier to that variable.

| Syntax | static variable type variable_name; |
|---|---|

For example :

```
static int number;
```

### 1.6.4 External

If a variable is called from functions defined in diferent files, the variable must be declared in all the functions that need to use this variable. Such declarations should be explicitly declared with the keyword extern. Declarations using the keyword extern do not allocate storage for the variables, but just defines the variable type and references an already defined variable. The actual declaration of the variable should be done only once. All other declarations should just be references. The syntax for defining an external variable outside any function is:

| Syntax | variable_type variable_name; |
|---|---|

The following example allocates storage for the variable number and has the type int.

```
int number;
```

To access this same variable in other files, the variables should be declared as :

| Syntax | extern variable_type variable_name; |
|---|---|

the following declaration defines the variable number as an int whose storage is declared somewhere else.

```
extern int number;
```

As external variables access the same storage, the value of the variable can be changed in any function that accesses the variable. Thus, the value of an external variable can be changed in unexpected places and it is difficult to keep track of the changes. Because of this, external variables should be used sparingly and avoided as much as possible.

## CHAPTER – 8

# 1. ARRAYS

An array is a blok of contiguous (adjoining) data items of the same type. Using an array enables you to group several data items, with common characteristics, under one name.

For example, you may want to store the names of 20 of your close friends. You could have 20 different variables such as friend1, friend2.... friend20, but this would be unwieldy; or, you can define an array called friends[20] to hold the same information in one structure.

In this chapter we will cover declaring, accessing and initializing oneand multi-dimensional arrays. We will also look at arrays as function arguments, and at strings.

### 8.1 Array Declaation

An array is composed of individual storage blocks called elements. The number of elements in the array determine the sizeof the array. For example, the array friends[20] has a size of 20, which means that it can hold up to 20 elements.

Arrays can be one-dimensional or can be multi-dimensional. In this section we will discuss one-dimensional arrays. Multi-dimensionsl arrays are covered in section 7.5.

One dimensional arrays are declared as follows :

| Syntax | data_type array_name[expression]; |
|---|---|

The following example shows the declaration of a one-dimensional array. Here f is anarray of floats which can hold up to 5 floating numbers.

```
float f[5];
```

In the following example, text is an array of characters which can hold a maximum of 90 characters.

```
char   text[90];
```

In the following example, a is an array of integers which can hold a maximum of 10 decimal integers. Another way of stating this is that a is an array of type int, and size 10.

```
int  a[10];
```

## 8.2 Accessing Array Elements

Each element of the array is referred to using a subscript called an index. It is important to note that in c, indexing starts from zero (0) onwards.

In figure 7.1, you have 20 elements ranging from 0 to 19. The address of the first friend is friends[0], the second friend is friends[1], and so on until th address of the 20th friend is friends[19].

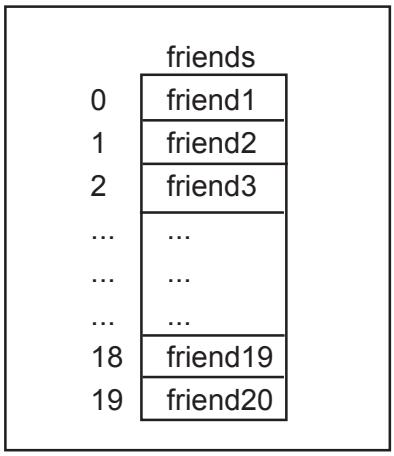



Figure 7.1 Physical Model of an Array

When performing operatinos using array elements (arithmeticl logical, relations, or bitwise), one an write a [i] to access the i-th element in a one-dimensional array, or a[i][i] to access the i-th row and j-th column element in a two-dimensional array.

## 8.3 Array Initialization

There are several different ways to initialize an array.

In the following example, the array (int a[10];) is initialized using a for loop.

```
for(i=0; i<10;i++)
a[i]=0;
```

In the following example, the two-dimensional array (int c[10][20];) is initialized using nested for loops.

```
for(i=0; i<10;i++)
   for(j=0; j<20; j++)
      c[i][j] = 0;
```

Arrays can also be initialized when they are delared as shown in the

following example

```
int a[5] = {0, 1, 2, 3, 4};
int b[2] [3] = {11, 22, 33, 21, 22,33};
int b[2] [3] = {11, 22, 33}, {21, 22, 33}};
int b[2] [3] = {
      {11, 22, 33},
       {21, 22, 33}
};
char name[12] = "Peter Smith";
char name[12] = {'P', 'e', 't', 'e', 'r', ' ', 's', 'm', 'i', 't', 'h', '\0'};
```

In the following example, the sum of the first three numbers is calculated and the result is stored in the 4th array element (a[3]).

```
a[3] = a[0]+a[1]+a[2];
```

The following program demonstrates the declaration, population, printing, and summing (accessing the elements of) an array.

```
/* ..................................................................................... */
/* Name : arrayex 1.c - demonstrates; declaration, filling, printing, summing     */
          (elements of) an array                                                    */
/* Date   : 5/29/97                                                                 */
/* Author : Prasad Putta                                                            */
/* ..................................................................................... */

#include<tdio.h>

#define N 5                                  /* Symbolic definition for size of the array*/
void main( )
{

        int  test_array[N];                                    /* Initialization of array */
        int  i, sum = 0;
        for (i=0 i<N; ++i)                                      /* Declaratino of array*/
             test_array[i] = 7+i*i;                /*the formula was choosen randomly*/
        for(i=0; i<N; ++i)                                          /*Printing the array*/
            printf("test_array[%d] = %d\n", i, test_array[i]);

        for (i=0; i<N; ++i)                                  /*Sum the array's elements*/
            sum + = tes_array[i];
        printf("\nsum = %d\n", sum);
        return;
}
```

The following program prompts you to enter your name, reads it into an array, an dprints it in the reverse order.

```
/* ..................................................................................... */
/*  Name : arrayex2.c - reversing a characer string (array) using a for loop      */
/*  Date : 5/29/97                                                                */
/*  Author : Prasad Putta                                                         */
/* ..................................................................................... */

#include <stdio.h>
void main ( )
{
        Char name [100] c;                                  /*Declaration of array*/
        int i;

                                  /* Prompting and reading the name into char array*/
        printf("Hi! What is your name?");
        for (i=0; (c = getchar( ) ) != \n'; ++i)
            name[i] = c;
        name [i] = '\0';

/*Printing the name in the correct order and in the reverse order*/
        printf("Nice to meet you, %s.\n", name);
        printf("Your name in the reverse order is;  ");
        for (--i; i>=0; --i)
             putchar(name[i]);

        return;
}
```

### 1.4 Arrays as Function Arguments

Suppose we declare an array int a[10]; and we want to send this array as a function argument. The function prototype for the function named comp will look like the following :

```
void comp(int b[ ] ); /*function prototype */
```

The function call inthe main( ) function will look like the following :

```
main (void)
{
int a[10];
.....
.....
.....
```

```
comp(a); /*this is a call to function comp( ) */
.....
.....
}
```

And the function definition will look like the following :

```
void comp(int b[ ])
{
.....
.....
}
```

You shuld note that pasing an array as a function argument will always result in a call by referece rather than a call by value.

### 8.5 Multi-Dimensional Arrays

So far we have touched briefly on multi-dimensional arrays. We will cover them in more depth in this section.

#### 8.5.1 Two-Dimensional Array

The following example shows the declaration of a two-dimensional array. In this example, a is a two-dimensional array of size 10*5, i.e. a matrix of size of * 5. It can be viewed as a matrix with 10 rows and 5 columns.

```
int a[10][5];
```

The first dimension defines the number of rows in the matrix and the second dimension defines the number of columns. To access an element in the i-th row and j-th column, you can write a[i][j].

In the following example, b is a two-dimensional array of size 15 with 5 rows (0-4) and 3 columns (0-2). Only characters can be stored in the array b.

```
char b[5][3];
```

The following program demonstrates the declaration, filling, and printing, of a two-dimensional array.

```
/* ..................................................................... */
/* Name : array2d.c - demonstrates : declaration, filling, printing, of a two  */
/*          dimensional array                                                   */
/* Date : 5/29/97                                                               */
/* Author : Pasad Putta                                                         */
/* ..................................................................... */
```

```
#include <stdio.h>

#define NROWS 3                          /*Symbolic definition for size of the array*/
#define NCOLS 4

void main ( )
{
        int a[NROWS] [NCOLS], i,j;                              /*Declaration of array*/

        for (i=0; i<NROWS; ++I)                                 /*Declaration of array */
            for (j=0; j<NCOLS; ++j)
             a[i][j]   =  i + j;                        /*the formula was chosen randomly*/
/* Printing the array */
        for  (i=0; i<NROWS; ++i){
             for (j = 0; j<NCOLS; ++j)
             a[i][j] = i + j;                           /*the formula was chosen randomly*/
/*Printing the array*/
        for (i = 0;p i<NROWS; ++i){
            for (j=0; j<NCOLS; ++j)
             printf("a[%d] [%d] = %d ", i, j, a[i][j] );
             printf("\n");
        }
        return;
}
```

### 8.5.2 Three Dimensional Array

The following example shows how a three-dimensional array is declared. In this example, a is a stack of 10 matrices of size 5 * 3. The total number of elements which can be stored in this array is 10*5*3 = 150. It can viewed as book with 10 pages and in each page, there is a matrix with 5 rows and 3 columns.

```
int   a[10][5][3];
```

## 8.6 Strings

A string is an array of characters terminated by the null characte(\0). The length of memory used to store a string is one more than the number of characters between the double quotes. Strings can be initialized in many ways as shown in the following examples.

In the following example, A is stored in str[0], B in str[1], C in str[2], and \0 in str[3]. The size of the array, 4, means that we cannot add any more characters into this array unless we change the array's limit, for example by writing str[number] where number > 4.

```
char str[4] = { 'A','B','C' } ;
```

As shown in the following example, strings of different lengths can be stored without sepcifying the size of an array. Individual characters within the array may be changed, but astr will continue to refer to the same storage.

```
char astr[ ] = "This is C Programming";
```

A string can also be defined using pointers. We will cover pointers in Chapter 8.

In the following example, 'a' can be considered a string. You can store a maximum of 10 characters starting from a[0] to a[9]. A null character should be the last character in a string. Keep in mind that /0 is a valid character and occupies memory.

```
char a[10];
```

In the following example, the statement reads a string into the array 'a'. Note that the address of operator (&) is not necessary to input a string.

```
scanf("%s",a);
```

Suppose the input is "HELLO". This string is stroed in a as shown below :

| H | E | L | L | O | \0 |
|---|---|---|---|---|---|

Note that, you may sometimes have problems reading strings using scanf(). It is recommended that you use either gets() or getchar() functions in a loop to read in a string.

In some programming languages, strings are treated as special datatypes, but in C they are treated as character arrays only. In spite of this, strings are treated differently compared to other types of data. There are special library functions defined to handle string manipulations.

### 8.6.1. String Functions

The C standard library provides a number of string manipulation functions. Generally you also use the string. h library file for string manipulatin functions. Table 7.1 lists a few of these functions.

| | |
|---|---|
| strcpy (a,b); | a and b are two strings, strcpy() stands for string copy and string b is copied into string a. |
| strncpy(a, b, n); | strncpy() is similar to strcpy() with a limitation that at most n characters are copied from the string b to the string a. |

| | |
|---|---|
| strcat(a, b); | strcat() function concatenates string b to the end of string a. |
| strnacat(a,b); | strncat() is similar to strcat() with a limitation that at most n characters from string b are concatenated to the end of the string a. |
| strlen(a); | returns the length of the string excluding the null character at the end. |

Table 7.1 String Maniulation Functions

There are other types of character functions which you may often come across in programming, such as converting characters from upper to lower case and vice-versa. Also, sometimes you may want to determine whether the character is a digit, a letter or a control character, and so on. In such cases you can use a special function library by including the ctype. h header file (for character functins). Tables 7.2 and 7.3 show some of these functions.

| **Function Name** | **Action** |
|---|---|
| tolower( ) | converts uppercase letter to lowercase |
| toupper( ) | converts lowercase letter to uppercase |

Table 7.2 Conversion Functions

| **Function Name** | **Function Returns Non-zero Value for :** |
|---|---|
| insalnum() | letter or digit |
| isalpha() | letter |
| isentrl() | control character |
| isdigit() | digit |
| isgraph() | printing character except space |
| islower() | lowercase letter |
| isprint() | printing character (including space) |
| ispunct() | printing character other than space, letter, or digit |
| isspace() | whitespace character |
| isupper() | whitespace character |
| isupper() | uppercase letter |
| isxdigir() | hexadecimal digit |

**Table 7.3 Functions for Testing Characters**

## Chapter - 8 Exercises

1. Write a program to prompt and read 10 grades (datatype float or double)into anarray. Compute the highest, lowest, and average of the 10 grades and print the 10 grades along with the results.
2. How do you initialize a 3 by 2 matrix (ex : array 1[3][2]) to zeros (each element of this matrix (array) should be zero).
3. What does the initialization statement : int array 1[3][2] = {0}; mean? Write a program which has this initialization statement and which will also print out the values of each of the elements of array1.
4. Modify the above progam to prompt the user for the values of the elements of array 1 and read them into array 1. This program should print out the array 1 (showing the values of each of the element of array1).
5. Write a program to prompt and read in values for two 2 x 3 matrices and one 3 x 1 matrix. Compute and printout the sum of first two matrices and the product of last two matrices along with the three matrices.
6. Write a program to read a series of letter grades terminated by a dollar sign. The program should count and print the number of A's, B's, C's and so on. The input looks like A A F B C F D B A C C B D F $
7. A magic square is a square array of integers such that every row, column, and diagonal will sum up to the same value as in :

| | | |
|---|---|---|
| 4 | 9 | 2 |
| 3 | 5 | 7 |
| 8 | 1 | 6 |

Write a program to prompt the user for the elements of the square in row-by-row order. The data for the example square will be : 4 9 2 3 5 7 8 6. The data should be stored in an array. Your program should check whether a square is a magic square by computing and comparing the sum of each row, each column, and the diagonals and display the result.

## CHAPTER – 9

# 1. POINTERS

A pointer is a variable that can store the address of another variable. In this chapter we will cover addressing and dereferencing operators, declaring a pointer and how pointers function with arrays and strings.

### 9.1 Address and Dereferencing Operators

#### 1.1.1 Address

The address of operator is &. Using this operator, the address of objects (i.e. int, float, char, double,...) can be obtained. Addresses are represented using the hexadecimal number system.

In the following example, the address of the variable C is stored in the variable ptr, and ptr is said to be pointing to c. The & unary operator is applied only to objects in memory.

```
ptr = &c;
```

#### 1.1.2 Dereferencing Operator

The dereferencing, or indirection operator is*. When * is applied to a pointer, it accesses the object that the pointer points to.

In the following example the address of variable a is stored in ptr (ptr is pointing to a). Then b is assigned the value which is pointed to by the pointer ptr (10). Then the value 90 is assigned to the location that ptr is pointing to, in this case the variable a.

```
int a = 10, b = 20;
int * ptr;
ptr = &a;
b = *ptr;
*ptr = 90;
```

Another important point to note here is that the variable ptr, just like any other variable, will have its own unique address.

### 9.2 Declaration

As shown in the syntax, a variable is capable of pointing to the memory location of a variable of type variable_type. variable_type can be character, float, double, etc.

```
Syntax      variable_type  *variable;
```

In the following example, a is a pointer to a character, but cannot hold a character. It is capable of holding the addresses of the memory locations where the character is stored.

```
char  *a;
```

The following example demonstrates how to declare a pointer to an integer. This pointer can hold the address of an integer variable. Such addressing is called indirect addressing.

```
int *ptr;
```

Note that the pointer variable should be of the same type as the data that it points to. For example, you use a pointer of type int to point to data of the type decimal integer. In the above example, the pointer ptr can point to any data item of the decimal integer type.

To clarify the difference between direct and indirect addressing, in the following example, a can hold a float point number -- direct addressing, but b can hold the address where the float point number is stored -- indirect addressing.

```
float  a, *b;
```

In the following example, ip is a pointer pointing to an integer and i is of type integer.

```
int *ip;
int i=10;
```

In the following example, the address of i(i.e. x 100C) is stored in the variable ip. & i is the address of i, i.e. x 100C. & ip is the address of ip, i.e. x 1000. *ip is the integer value that it is pointing to, i.e. 10.

```
ip = &;
```

In the following example, this type of assignment is only allowed in the declaration and not elsewhere.

```
char c = 'Q';
char *char_pointer = &c;
```

Otherwise, you have the declaration and an assignment statement;

```
char *char_pointer;
char_pointer = &c;
```

Also note that, *char_pointer = c; is incorrect and will result in an error.

**Exercise :** Go through the following programs and write down what you think the output will be on a piece of paper. Write the output exactly as you expect to see it on the monitor once the program is executed, including spaces. Then type in the program, execute it and compare your output.

The following program aids in understanding the declaration and use of pointers:

```
/* ..................................................................... */
/*  Name : pointer 1.c - simple example showing the use of pointers     */
/*  Date : 5/29/97                                                      */
/*  Author : Prasad Putta                                               */
/* ..................................................................... */
#include <stdio.h>
/* ..................................................................... */
/* main() function                                                      */
/* ..................................................................... */
void main()
{
                                                          /* Declarations*/
        int num, *ptr;
                                                /* Assigning an address to a pointer */
        ptr = &num;                  /*ptr points to variable num i.e., variable ptr is
                                                    assigned the address of the num*/
        num = 10;
        printf("Num: %d, Ptr: %d\n", num, *ptr);
        *ptr = 20;
        printf("Num: %d, Ptr : %d\n", num, *ptr);
        printf("Num [%04x]  Ptr [%04x]   Ptr  %04x\n", &num, &ptr, ptr);

        return;
}
```

The following program aids in understanding the declaration and the use of pointers :

```
/*       */
/*  Name : pointer2.c - simple example showing the use of pointers     */
/*  Date : 5/29/97
/*  Author : Prasad Putta                                               */
/*       */
#include <stdio.h>
/* ..................................................................... */
/*  main() function                                                     */
/* ..................................................................... */
void main()
```

```
{
        int  x,  count = 10 *int_ptr;                                   /*Declarations*/
        int_ptr = &count;                          /* Assigning an address to a pointer */
        x = *int_ptr;
        printf("count = %d,  x = %d\n", count, x);
        printf("count [%p] %d\n", &count, count);
        printf("     x[%p] %d\n", &x, x);
        printf("int p[%p] %p  (ip: %d)\n", &int_ptr, *int_ptr);

        return;
}
```

The following program demonstrates functions, all by reference, and pointers:

```
/* ..................................................................................... */
/*  Name : funeptr1.c- simple example showing the use of pointers and functions */
/*  Date : 5/29/97                                                                     */
/*  Author : Prasad Putta                                                              */
/* ..................................................................................... */
#include <stdio.h>
/* ..................................................................................... */
/*  swap_em function                                                                   */
/*  This program contains the function definition above main and hence the prototype */
/*  is eliminated                                                                      */
void swap_em(int *x, int*y)
{
        int  temp;
        temp = *x;
        *x = *y;
        *y = temp;

        return;
}
/* ..................................................................................... */
/*  swap function                                                                      */
/*  Here also the prototype is eliminated as the function is above main( )            */
/* ..................................................................................... */
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y;
          y = temp;
        return;
}
/* ..................................................................................... */
```

```
/*  main( ) function                                                    */
/* ..................................................................... */

void main( )
{
        int num1 = 10,                                          /*Declarations*/
            num2 = 20;

        swap(num1, num2);
        printf("Num1: %d, Num2: %d\n", num1, num2);

        swap_em(&num1, &num2);
        printf("Num1: %d, Num2: %d\n", num1, num2);
        swap_em(&num1, &num2);
        printf("Num1 : %d, Num2: %d\n", num1, num2);

        return;
}
```

### 9.3 Pointers and Arrays

Array subscripts can be represented in pointer form. Any operation with array subscription can also be done with pointes. Operations using pointers are faster than operations using arrays.

In the following example, a is an array of 10 integers. All array elements are stored in consecutive memory locations named a[0], a[1], a[2].... a[9]. the notationa[i] refers to the i-th element in the array a.

```
int  a[10];
```

In the following example, the address of a[0] element is stored in the variable ptr.

```
ptr = &a[0];
or
ptr = a;
```

In the following example, x is given the value from a[0], the first element of the array.

```
x = *ptr;
```

The following example shows how to access the i-th element in the array a using a pointer. In this example, a[5] can also be accessed as *(ptr + 5).

```
*(ptr +i).
```

Pointers must point to variables of the same type. For example, a pointer to a float type cannot be used to save the address of an integer variable.

### 1.4 Pointers and Strings

Consider the following declaration :

```
char astr[ ] ="This is C Programming";
```

This can also be defined by using pointers as follows :

```
char *pstr = "This is C Programming";
```

The pointer pstr is initialized to point to a string constant. The pointer may subsequently be modified to point elsewhere, but the result is undefined if we try to modify the string contents.

**Exercise :** Go through the following porgram and write down what you think the output will be on a piece of paper. Write the output exactly as you expect to see it on the monitor once the program is executed, including spaces. Then type in the program, execute it and compare your output.

The following progam demonstrates the use of strings and pointers. Usually, with string manipulations, you also use the #include<string.h> header file.

```
/* ..... */

/*  Name : funcptr1.c - simple example showing the use of pointes and functins */
/*  Date : 5/29/97                                                              */
/*  Author : Prasad Putta                                                       */
/* ..... */

#include <stdio.h>
#include <string.h>
/* ..... */
/* display function                                                             */
/* This program contains the function definition above main and hene the prototype */
/* ..... */

void display (char *ptr)
{
printf("%s\n", ptr);
return;
}
/* ..... */
/* main() function                                                              */
/* ..... */
void main( )
```

```
{

    char name[30], *ptr;                                    /*Declarations*/
    ptr = name;
    strepy(name, "Compulearn");

    display (name);
    display(ptr);
    return;
}
```

## CHAPTER – 10

# 1. DYNAMIC MEMORY ALLOCATION

### 10.1 Overview

Storage is set aside for variables when they are declared. This type of memory allocation is called static memory allocation. Static memory is allocated when the code is compiled.

When an array is declared, a block of memory is assigned to the array. To declarean array you need to know the size of the array at the time of coding. If the size of the array will be known only during the program execution, then it is not possible to declare a static array. In this case, memory should be allocated during program execution. This type of memory allocation is called dynamic memory allocation. Routines available for dynamic memory allocation are malloc(), calloc(), and realloc(). All these routines return a pointer to memory. The pointer is of the void type and should be cast to the object allocaated's type.

### 10.2 Allocating Memory

**Malloc () :** The term malloc comes from "memory allocation." The storage allocated by malloc() is not initialized and starts with garbage values. In the following syntax, size is the total size in bytes needed.

```
Syntax    void *malloc(size_t size);
```

The following is an example of how a call to malloc() should look. ptr is a pointer to integer and N is the number of integers to be allocated. The size of operator gives the size of the type in bytes. The pointer returned by malloc() should be cast to the appropriate object type.

```
ptr = (int*) malloc (sizeof() *N);
```

**calloc():** The term calloc comes from "contiguous allocation." The storage allocaated by calloc() is automatically initialized to zero. In the following syntax, N is the number of objects of size obj_size.

```
Syntax   void *calloc(size_t  N, size_t  obj_size);
```

A call to calloc() is similar to the call to malloc() and is as follows :

```
ptr = ( int*) calloc (N, size);
```

**Realloc() :** In the following ssyntax, p is a pointer to dynamic memory and size is the new size needed.

```
Syntax  void * realloc (void *p, size_t size);
```

The following is an example of how a call to realloc() should look:

```
new_ptr = (int *) realloc ( (void* ) ptr, M);
```

In this example, ptr is a pointer to dynamic memory and M is the new size to which ptr should be changed. The contents of the memory ptr is unchanged. realloc() is used to change the size of the memory pointed to by the pointer. It is used when more or less memory is needed than was originally allocated.

### 10.3 Deallocation

**free ( ) :** Dynamically allocated memory should be freed befoe exiting the routine using the routine free.

```
Syntax    void free (void * ptr);
```

## CHAPTER – 11

# 1. STRUCTURES

As we saw in Chapter 7, arrays provide a way of grouping items of the same type together. When items of different types are to be grouped into one logical unit, arrays cannot be used. For such situations, C provides a special data structure known as a structure. A structure is a datatype that enables the grouping of different types of data items under one name. The data items may or may not have the same type.

Structures help to organize complicated data. For example, if you want to develop a phone book, you need to have a number of variables to save things like name, street, city, etc. In this situation all these variables have a logical association with each other, but that associatin is not visible from looking at the declarations of these variables. Structures groups these variables into one logical unit. In this chapter we will discuss declaring and accessing structures, arrays of structures, pointers to structures, and ur, ions.

### 11.1 Declaration of A Structure

The keyword for defining structures is struct. The following example defines a structure to hold a person's address. In this example defines a structure to hold a person's address. In this example, address is called the structure tag and is optional. The structure tag gives the structure a name. Each item of the structure (name, street, city and zip) is called a membr of the structure.

```
struct address
{
char name[40];
char street[100];
char city[40];
long zip;
}'
```

### 11.2 Declaring a Structure Variable

You can declare variables of the structure type by using a structure tag. The following example declares a variable called person of type address, which is a structure.

```
Syntax    struct address personl;
```

You can also declare the variable person 1 as shown in the following example. In such declarations the structure tag can be omitted.

```
struct address {
        char name[40];
        char street[100];
        char city[40];
        long zip;
        }person1;
```

### 11.3 Accessing Members of Structure

The declarations within the structure block (braces) are called members of the structure. To refer to a member of a structure, a dot(.) operator should be used.

```
Syntax   Structure_varriable.Member
```

For example, to refer to the name field of he structure variable person1, you would write :

```
person1.name
```

Or, to refer to the zip field of the structure variable person1, you would write :

```
personal.zip
```

### 11.4 Arrays of Structures

#### 11.4.1 Declaration

An array of structures is declaredsimilarly to the way an array of integers is declared.

```
struct tag(name) variable [size];
or
struct tag (name) {
        type variable1;
        type variabla2;
        type variable3;
        }array variable[size of array];
```

In the following example, all the locations from a[0] to a [9] can store integer values.

```
int a[10];
```

Similarly, in the following example, all the locations from person[0] to person[99] can store the structure of type address.

```
struct address person[100]
```

### 11.4.2 Accessing Members of Array Structures

In the following syntax, we access the field of an i-th element of array-variable.

```
Syntax    array_variable[i] field
```

In the following example, we access the name field of the 51 st person (51 because the array starts from 0 not 1).

```
Peerson[50].name
```

## 11.5 Pointers to Structures

Pointers to structures can also be declared and the syntax is similar to any other pointer declaration. Here, variable is a pointer pointing to a structure of type structure-name.

```
Syntax    Struct Structure_name *variable;
```

The following example declares p to be a pointer to a structure called address.

```
struct address*p.
```

### 11.5.1 Accessing Structure Members Through Pointer Variables

To refer to structure members through pointer variables, the arrow operator(->) or the dot operator(.) can be used. It returns the field value pointed to by the pointer_variable.

```
Syntax    *pointer_variable.field
          or
          pointer_variable_field
```

For example; the name field of the structure address using pointer p can also be accessed using either the arrow operator or the dot operator as shown below:

```
(*p) name
p-> name
```

## 11.6 UNIONS

Unions have the same syntax as structures but with one basic difference. In a structure, all members have their own memory locations; whereas in a union, all members share the same slot of memory, which is big enough to hold the biggest type of member. As all the members of a union share the same memory slot, only one member can be used at any time.

The keyword for defining unions is union.

```
Syntax    union union-tag{
          int 1;
          float val
          char status;

          } union_variables;
```

Similar to a structure definition, the union_tag and union_variables are optional when a union is defined. Only the last filled variable can be accessed and it is the users responsibility to keep track of the last variable filled.

To access a union member, the dot operator should be used if accessing the member through a pointer variable. This is similar to accessing members of a structure. Unions should be used when a particular item can take different types at different instances and only one value of the item is used at any given time.

# COMPUTER FUNDAMENTALS
# AND
# PROGRAMMING IN 'C'

# CONTENTS

H H H